TURING 图灵程序设计丛书

Understanding and Using C Pointers

C/C++程序员进阶必备，透彻理解指针与内存管理

深入理解C指针



[美] *Richard Reese* 著

陈晓亮 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

陈晓亮

美团网iOS工程师，长期关注Linux、iOS、C、Objective-C，推崇C语言，认为程序员的C功底会直接影响Objective-C代码质量。他重视用户体验，喜欢干净的代码，业余时间喜欢读书，经常与大家探讨技术问题，译有《iOS 6编程实战》一书。

 图灵程序设计丛书

深入理解C指针

Understanding and Using C Pointers

[美] Richard Reese 著
陈晓亮 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

深入理解C指针 / （美）里斯（Reese,R.）著；陈晓
亮译. -- 北京：人民邮电出版社，2014.2（2016.12重印）
（图灵程序设计丛书）
书名原文：Understanding and using C pointers
ISBN 978-7-115-34448-9

Ⅰ. ①深… Ⅱ. ①里… ②陈… Ⅲ. ①C语言－程序设
计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第008761号

内 容 提 要

指针是 C 语言的一项核心特性，对于指针的掌握程度是区分 C 语言新手与老手的重要标志。本书专门研究指针，旨在提供比其他图书更全面和深入的 C 指针和内存管理知识。全书巨细靡遗地为读者展现了 C 语言编程中最重要的话题：C 的动态内存管理、指针和函数、指针和数组、指针和字符串、指针和结构体。作者还专门用一章篇幅讲解了安全问题和指针误用，让读者对指针的认识又深入一层。全书每章都穿插了使用指针的注意事项和潜在陷阱，及时贴心地提示读者在编程中避开此类问题。

本书适合 C 和 C++ 程序员和开发人员阅读，也可作为计算机专业学生学习 C 语言的参考图书。

◆ 著 [美] Richard Reese
译 陈晓亮
责任编辑 刘美英
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：12.75
字数：243千字 2014年 2 月第 1 版
印数：4 301－4 600册 2016年 12 月北京第 3 次印刷

著作权合同登记号 图字：01-2014-0409号

定价：45.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

前言

C 是一门重要的编程语言，一直以来备受关注。C 语言的核心是指针，这门语言的灵活性和超长之处很大一部分都源于指针。指针提供了动态操控内存的机制，强化了对数据结构的支持，且实现了访问硬件的功能。不过，指针的这种能力和灵活性是有代价的，它很难掌握。

本书的不同之处

市面上已经出版了大量关于 C 的图书，这些书一般会讲到语言的方方面面，但是对于指针的讲解却翻来覆去总是那点东西，很少涉及指针基本知识以外的内容。这些书大部分只是粗略地提一下内存管理，其实内存管理技术很重要，会牵涉栈和堆。不讨论这些内容，读者就只能粗浅地理解指针。栈和堆是分别用来支持函数和动态内存分配的内存区域。

指针非常复杂，值得我们深入讨论。本书专门研究指针，旨在让读者对指针的理解更加深入。这种理解在某种程度上需要读者会使用程序栈和堆，且会在相关上下文背景下使用指针。人们对不同领域知识的理解程度不同，有大体粗略了解，也有深入透彻的理解。要想对 C 语言的理解达到更高的层次，就必须有坚实的指针和内存管理基础。

学习方法

编程其实就是操控数据，数据一般位于内存中。因此如果能更好地理解 C 如何管理内存，就能对程序的工作原理洞若观火，从而使编程能力更上一层楼。知道 `malloc` 是从堆上分配内存是一回事，理解内存分配意味着什么则是另一回事。如果我们分配一个逻辑长度是 45 字节的结构体，可能会惊讶地发现实际上分配的空间

往往大于 45 字节，而且分配的内存可能会碎片化。

调用函数过程中系统会创建一个栈帧并将其推到程序栈上。理解栈帧和程序栈就可以弄清楚传值和传址的概念。尽管栈帧和指针没有直接联系，但是理解栈帧有助于理解递归的工作原理。

为方便理解指针和内存管理技术，我们会讲到不同的内存模型，有简单的线性内存布局，也有更复杂的针对某个特定示例说明程序栈和堆状态的图例。显示在屏幕上或是印刷在书中的代码只是动态程序的一种静态表示，这种表示的抽象本质是理解程序行为最大的绊脚石。内存模型对清除这些绊脚石非常有用。

目标读者

C 语言是块结构的语言，其过程式编程方法在 C++ 和 Java 等很多现代语言中也被采用。这些语言都要用到程序栈和堆，也都会用到指针，只不过通常用引用来包装。阅读本书要求你对 C 有最基本的了解。如果你正在学 C，那么本书能为你提供比其他书更全面的指针和内存的知识，能扩大你的 C 知识面，也能让你发现自己在 C 上的薄弱环节。如果你是有经验的 C 或者 C++ 程序员，本书能帮你填补对 C 语言理解的空白，也能强化你对这两门语言工作原理的理解，从而让你成长为更优秀的程序员。如果你是 C# 或者 Java 开发者，本书能让你更好地理解 C，也能让你更加透彻地认识面向对象语言如何处理栈和堆。

本书结构

本书按照数组、结构体、函数等传统主题组织。不过，所有章节关注的焦点都在于指针的使用和内存管理。比如说，我们讲到了如何给函数传递指针和从函数返回指针，也讲到了指针在栈帧中的使用以及指针如何引用堆中的内存。

- 第1章，认识指针
  这一章为非专业人士和对指针比较陌生的读者介绍指针的基础知识，包括指针操作符和如何声明不同类型的指针（比如常量指针、函数指针），以及如何使用 `NULL` 及相关变体。这对内存的分配和使用方式有很大的影响。

- 第2章，C的动态内存管理
  主要介绍标准的内存分配函数和回收内存的相关技术。能否及时回收内存对大部分应用程序来说都至关重要，做不到这一点就会导致内存泄漏和迷途指针。我们也讲到了垃圾收集和异常处理函数等回收技术。

- 第3章，指针和函数
  函数是组成应用程序代码的基石，不过给函数传递数据和从函数返回数据可能让开发新手迷糊。这一章介绍传递数据的技术，以及用指针返回数据时的常见陷阱。另外还会详细讲解函数指针，这种指针为应用程序提供了另一层控制和灵活性。
- 第4章，指针和数组
  尽管指针表示法和数组表示法不能完全互换，但两者紧密相关。这一章介绍一维和多维数组及其与指针的配合使用，还特别针对不同的内存模型解释如何传递数组，以及用连续或是非连续的方式动态分配数组时的一些常见问题。
- 第5章，指针和字符串
  字符串是很多应用程序的重要组成部分，这一章介绍字符串的基本知识，以及如何用指针操控字符串。字面量池及其对指针的影响也是很多人忽视的一个 C 特性。这一章会用丰富的图例来解释和说明这个主题。
- 第6章，指针和结构体
  结构体提供了一种排序和操控数据的有用方式，指针则通过在结构体的创建方式上提供更多的灵活性进一步强化了结构体的用途。这一章介绍结构体的基本知识及其与内存分配和指针的关系，接着举例说明如何在几种数据结构中使用结构体。
- 第7章，安全问题和指针误用
  指针很强大，但也可能造成很多安全问题。这一章研究缓冲区溢出的基本问题以及相关的指针问题，也会讲到避免这类问题的技术。
- 第8章，其他重要内容
  最后一章介绍一些比较零散但重要的指针技术和问题。尽管 C 不是面向对象语言，但是很多面向对象的编程思想可以融入 C 程序，包括多态。还会讲到在多线程环境中使用指针的要点，以及 `restrict` 关键字的意义和用法。

小结

本书旨在为读者提供比其他相关图书更深入的关于指针用法的知识，展示从指针的核心用法到罕见用法的例子，同时也指出常见的指针问题。

排版约定

本书使用的排版约定如下。

- 楷体

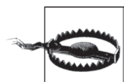
  表示新术语。

- 等宽字体

  表示程序代码，也用于正文中表示程序中使用的变量、函数名、命令行代码、数据类型、环境变量、语句和关键字等代码文本。



这个图标表示提示、建议或是一般注意事项。



这个图标表示警告。

使用代码

本书就是要帮读者解决实际问题的。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。因此，用本书中的几段代码写成一个程序不用向我们申请许可。但是销售或者分发 O'Reilly 图书随附的代码光盘则必须事先获得授权。引用书中的代码来回答问题也无需我们授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版商和书号，例如：*Understanding and Using C Pointers* (O'Reilly). Copyright 2013 Richard Reese, Ph.D. 978-1-449-34418-4。

如果还有其他使用代码的情形需要与我们沟通，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online



Safari Books Online（www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O’Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和发现的问题发给出版社。

美国：

　　O’Reilly Media, Inc.
　　1005 Gravenstein Highway North
　　Sebastopol, CA 95472

中国：

　　北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
　　奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

　　http://oreil.ly/Understand_Use_CPointers

对于本书的评论和技术性问题，请发送电子邮件到：

　　bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

　　http://www.oreilly.com

我们在 Facebook 的地址如下：

　　http://facebook.com/oreilly

请关注我们的 Twitter 动态：

　　http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

　　http://www.youtube.com/oreillymedia

第1章 认识指针

C 程序员新手和老手的一大差别就在于是否对指针有深刻理解，能否高效利用指针。指针在 C 语言中随处可见，也提供了极大的灵活性。指针为动态内存分配提供了重要支持，与数组表示法紧密相关，指向函数的指针也为程序中的流控制提供了更多的选择。

一直以来，指针都是学习 C 语言的最大障碍。指针的基本概念很简单，就是一个存放内存地址的变量。然而，当我们开始应用指针操作符并试图看懂那些令人眼花缭乱的符号时，指针就开始变得复杂了。但情况并非总是如此，如果我们从简单的知识入手，打好扎实的基础，那么掌握指针的高级应用并不难。

理解指针的关键在于理解 C 程序如何管理内存。归根结底，指针包含的就是内存地址。不理解组织和管理内存的方式，就很难理解指针的工作方式。为此，只要对解释指针的原理有帮助，我们就会说明内存的组织方式。牢牢掌握了内存及其组织方式，理解指针就会容易很多。

本章简要介绍指针、指针操作符以及指针如何与内存相互作用。1.1 节研究如何声明指针、基本的指针操作符和 `null` 的概念。C 支持好几种不同类型的 `null`，所以仔细研究 `null` 会对我们有所启发。

1.2 节将细致地介绍几种不同的内存模型。毫无疑问，我们在使用 C 的过程中肯定会遇到各种内存模型。特定编译器和操作系统下的内存模型会影响指针的使用方式。我们也将仔细研究跟指针和内存模型有关的几种预定义类型。

1.3 节会深入探讨指针操作符，包括指针的算术运算和比较。1.4 节探究常量和指针。众多的声明组合提供了有趣通常也很有用的方法。

无论你是 C 程序员新手还是老手，本书都能帮助你深入理解指针，填补你知识结构中的空白。老手可以挑选感兴趣的主题，新手还是按部就班为好。

1.1 指针和内存

C 程序在编译后，会以三种形式使用内存。

- *静态/全局内存*
  静态声明的变量分配在这里，全局变量也使用这部分内存。这些变量在程序开始运行时分配，直到程序终止才消失。所有函数都能访问全局变量，静态变量的作用域则局限在定义它们的函数内部。

- *自动内存*
  这些变量在函数内部声明，并且在函数被调用时才创建。它们的作用域局限于函数内部，而且生命周期限制在函数的执行时间内。

- *动态内存*
  内存分配在堆上，可以根据需要释放，而且直到释放才消失。指针引用分配的内存，作用域局限于引用内存的指针，这是第 2 章的重点。

表 1-1 总结了这些内存区域中用到的变量的作用域和生命周期。

表1-1：不同内存中变量的作用域和生命周期

	作　用　域	生命周期
全局内存	整个文件	应用程序的生命周期
静态内存	声明它的函数内部	应用程序的生命周期
自动内存（局部内存）	声明它的函数内部	限制在函数执行时间内
动态内存	由引用该内存的指针决定	直到内存释放

理解这些内存类型可以更好地理解指针。大部分指针用来操作内存中的数据，因此理解内存的分区和组织方式有助于我们弄清楚指针如何操作内存。

指针变量包含内存中别的变量、对象或函数的地址。对象就是内存分配函数（比如 `malloc`）分配的内存。指针通常根据所指的数据类型来声明。对象可以是任何 C 数据类型，如整数、字符、字符串或结构体。然而，指针本身并没有包含所引用数据的类型信息，指针只包含地址。

1.1.1 为什么要精通指针

指针有几种用途，包括：

- 写出快速高效的代码；
- 为解决很多类问题提供方便的途径；
- 支持动态内存分配；
- 使表达式变得紧凑和简洁；
- 提供用指针传递数据结构的能力而不会带来庞大的开销；
- 保护作为参数传递给函数的数据。

用指针可以写出快速高效的代码是因为指针更接近硬件。也就是说，编译器可以更容易地把操作翻译成机器码。指针附带的开销一般不像别的操作符那样大。

很多数据结构用指针更容易实现，比如链表可以用数组实现，也可以用指针实现。然而，指针更容易使用，也能直接映射到下一个或上一个链接。用数组实现需要用到数组下标，不直观，也没有指针灵活。

图 1-1 比较形象地展示了用数组和指针实现员工链表时的情形。图中左边用了数组，head 变量表明链表的第一个元素在数组下标 10 的位置，每一个数组元素都包含表示员工的数据结构。结构的 next 字段存放下一个员工在数组中的下标。灰底的元素表示未使用。

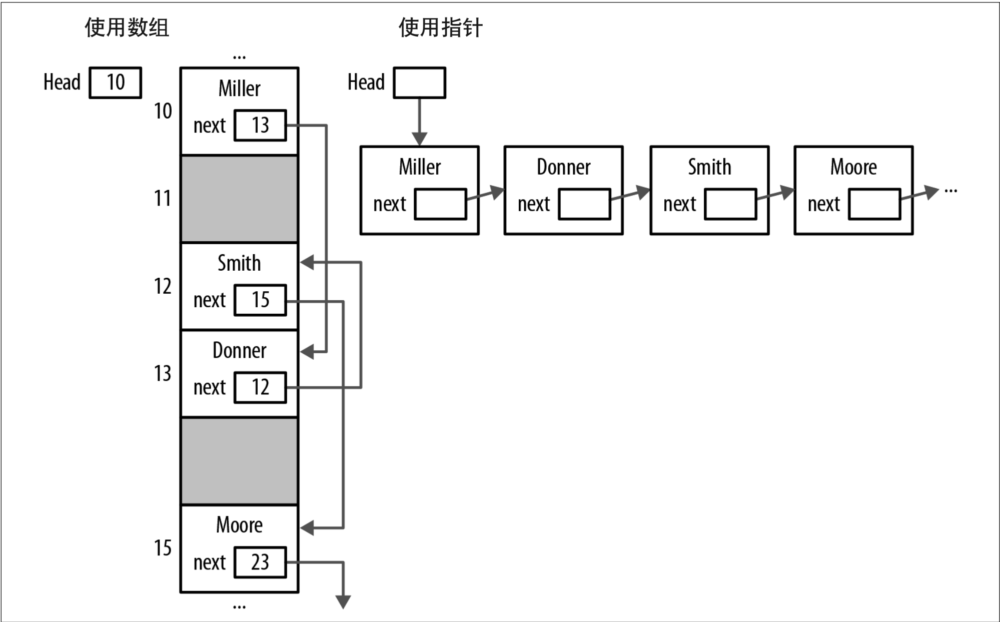


图 1-1：链表的数组形式和指针形式

右边显示了用指针实现的等价形式。head 变量存放指向第一个员工节点的指针。每个节点存放员工数据和指向链表中下一个节点的指针。

指针形式不仅更清晰，也更灵活。通常创建数组时需要知道数组的长度，这样就会限制链表所能容纳的元素数量。使用指针没有这个限制，因为新节点可以根据需要动态分配。

C 的动态内存分配实际上就是通过使用指针实现的。malloc 和 free 函数分别用来分配和释放动态内存。动态内存分配可以实现变长数组和数据结构（如链表和队列）。不过，新的 C11 标准也支持变长数组了。

紧凑的表达式有很强的表达能力，但也比较晦涩，因为很多程序员并不能完全理解指针表示法。紧凑的表达式应该用来满足特定的需要，而不是为了晦涩而晦涩。比如说，下面的代码用了两个不同的 printf 函数调用来打印 names 的第二个元素的第三个字符。如果对指针的这种用法感到困惑，不用担心，我们会在 1.1.6 节中详细介绍解引（dereference）的工作原理。尽管两种方式都会显示字母 n，但是数组表示法更简单。

```
char *names[] = {"Miller","Jones","Anderson"};
printf("%c\n",*(*(names+1)+2));
printf("%c\n",names[1][2]);
```

指针是创建和加强应用的强大工具，不利之处则是使用指针过程中可能会发生很多问题，比如：

- 访问数组和其他数据结构时越界；
- 自动变量消失后被引用；
- 堆上分配的内存释放后被引用；
- 内存分配之前解引指针。

我们会在第 7 章深入研究这几类问题。

指针的语法和语义在 C 规范（http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf）中讲得很清楚了，但还是有一些情况下规范没有明确定义指针的行为。这类情况下，指针的行为定义为如下之一。

- 实现定义

  有具体的实现，并且有文档描述。实现定义行为的一个例子就是当整数做右移操作时如何补充符号位。

- *未确定*

  有某种实现，但是没有文档描述。未确定行为的一个例子是当 `malloc` 函数的参数为 0 时所分配的内存大小。在 CERT Secure Coding Appendix DD 有一个未确定行为的列表（参见 https://www.securecoding.cert.org/confluence/display/seccode/DD.+Unspecified+Behavior）。

- *未定义*

  没有规定，任何事情都有可能发生。这种行为的一个例子是被 `free` 函数释放的指针的值。CERT Secure Coding Appendix CC 有一个未定义行为的列表（参见 https://www.securecoding. cert.org/confluence/display/seccode/CC.+Undefined+Behavior）。

有时候还会有语言环境相关的行为，这些行为一般由编译器厂商的文档说明。提供语言环境相关的行为能够为编译器作者生成更高效的代码提供更多空间。

1.1.2　声明指针

通过在数据类型后面跟星号，再加上指针变量的名字可以声明指针。下面的例子声明了一个整数和一个整数指针：

```
int num;
int *pi;
```

星号两边的空白符无关紧要，下面的声明都是等价的：

```
int* pi;
int * pi;
int *pi;
int*pi;
```



空白符的使用是个人喜好。

星号将变量声明为指针。这是一个重载过的符号，因为它也用在乘法和解引指针上。

对于以上声明，图 1-2 说明了典型的内存分配是什么样的。三个方框表示三个内存单元，每个方框左边的数字是地址，地址旁边的名字是持有这个地址的变量，这里的地址 100 只是为了说明原理。就这个问题来说，指针或者其他变量的实际地址通常是未知的，而且在大部分的应用程序中，这个值也没什么用。三个点表示未初始化的内存。

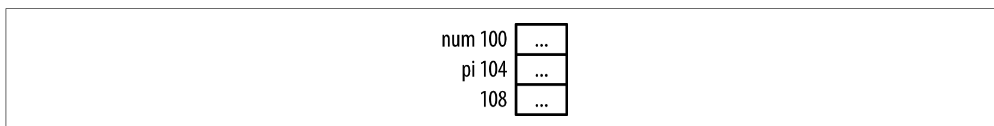


图 1-2：内存图解

指向未初始化的内存的指针可能会产生问题。如果将这种指针解引，指针的内容可能并不是一个合法的地址，就算是合法地址，那个地址也可能没有包含合法的数据。程序没有权限访问不合法地址，否则在大部分平台上会造成程序终止，这是很严重的，会造成一系列问题，第 7 章将讨论这些问题。

变量 `num` 和 `pi` 分别位于地址 100 和 104。假设这两个变量都占据 4 字节空间，就像 1.2 节中所说，实际的长度取决于系统配置。除非特别指明，我们所有的例子都使用 4 字节长的整数。



本书用 100 这样的地址来解释指针如何工作，这样会简化例子。当你运行示例代码时会得到不同的地址，而且这些地址甚至在同一个程序运行几次的时候都可能变化。

记住这几点：

- `pi` 的内容最终应该赋值为一个整数变量的地址；
- 这些变量没有被初始化，所以包含的是垃圾数据；
- 指针的实现中没有内部信息表明自己指向的是什么类型的数据或者内容是否合法；
- 不过，指针有类型，而且如果没有正确使用，编译器会频繁抱怨。



说到垃圾，我们是指分配的内存中可能包含任何数据。当内存刚分配时不会被清理，之前的内容可能是任何东西。如果之前的内容是一个浮点数，那把它当成一个整数就没什么用。就算确实包含了整数，也不大可能是正确的整数。所以我们说内容是垃圾。

尽管不经过初始化就可以使用指针，但只有初始化后，指针才会正常工作。

1.1.3　如何阅读声明

现在介绍一种阅读指针声明的方法，这个方法会让指针更容易理解，那就是：倒过来读。尽管我们还没讲到指向常量的指针，但可以先看看它的声明：

```
const int *pci;
```

倒过来读可以让我们一点点理解这个声明（见图 1-3）。

1. pci 是一个变量	const int ***pci;**
2. pci 是一个指针变量	const int ***pci;**
3. pci 是一个指向整数的指针变量	const **int *pci;**
4. pci 是一个指向整数常量的指针变量	**const int *pci;**

图 1-3：倒过来的声明

很多程序员都发现倒过来读声明就没那么复杂了。



遇到复杂的指针表达式时，画一张图，我们在很多例子中就是这样做的。

1.1.4 地址操作符

地址操作符 & 会返回操作数的地址。我们可以用这个操作符来初始化 pi 指针，如下所示：

```
num = 0;
pi = &num;
```

num 变量设置为 0，而 pi 设置为指向 num 的地址，如图 1-4 所示。

num 100	0
pi 104	100
108	...

图 1-4：内存赋值

可以在声明变量 pi 的同时把它初始化为 num 的地址，如下所示：

```
int num;
int *pi = &num;
```

有了以上声明，下面的语句在大部分编译器中都会报语法错误：

```
num = 0;
pi = num;
```

错误看起来可能是这样的：

```
error: invalid conversion from 'int' to 'int*'
```

pi 变量的类型是整数指针，而 num 的类型是整数。这个错误消息是说整数不能转换

为指向整数类型的指针。



把整数赋值给指针一般都会导致警告或错误。

指针和整数不一样。在大部分机器上，可能两者都是存储为相同字节数的数字，但它们不一样。不过，也可以把整数转换为指向整数的指针：

```
pi = (int *)num;
```

这样不会产生语法错误。不过运行起来后，程序可能会因为试图解引地址 0 处的值而非正常退出。在大部分操作系统中，在程序中使用地址 0 是不合法的。我们会在 1.1.8 节中详细讨论这个问题。



尽快初始化指针是一个好习惯，如下所示：

```
int num;
int *pi;
pi = &num;
```

1.1.5　打印指针的值

我们实际使用的变量几乎不可能有 100 或 104 这样的地址。不过，变量的地址可以通过打印来确定，如下所示：

```
int num = 0;
int *pi = &num;

printf("Address of num: %d Value: %d\n",&num, num);
printf("Address of pi: %d Value: %d\n",&pi, pi);
```

运行后，会得到下面的输出。在这个例子中我们用了真实的地址，你的地址可能会不一样：

```
Address of num: 4520836  Value: 0
Address of pi: 4520824  Value: 4520836
```

`printf` 函数还有其他几种格式说明符在打印指针的值时比较有用，如表 1-2 所示。

表1-2：格式说明符

格式说明符	含　　义
%x	将值显示为十六进制数
%o	将值显示为八进制数
%p	将值显示为实现专用的格式，通常是十六进制数

这些说明符的用法如下：

```
printf("Address of pi: %d Value: %d\n",&pi, pi);
printf("Address of pi: %x Value: %x\n",&pi, pi);
printf("Address of pi: %o Value: %o\n",&pi, pi);
printf("Address of pi: %p Value: %p\n",&pi, pi);
```

这样就会显示 pi 的地址和内容，如下所示。在这个例子中，pi 持有 num 的地址：

```
Address of pi: 4520824  Value: 4520836
Address of pi: 44fb78  Value: 44fb84
Address of pi: 21175570  Value: 21175604
Address of pi: 0044FB78  Value: 0044FB84
```

%p 和 %x 的不同之处在于：%p 一般会把数字显示为十六进制大写。如果没有特别说明，我们用 %p 作为地址的说明符。

在不同的平台上用一致的方式显示指针的值比较困难。一种方法是把指针转换为 void 指针，然后用 %p 格式说明符来显示，如下：

```
printf("Value of pi: %p\n", (void*)pi);
```

void 指针会在 1.1.8 节的“void 指针”中解释。为了保证示例简单，我们会用 %p 说明符，而不把地址转换为 void 指针。

虚拟内存和指针

让打印地址变得更为复杂的是，在虚拟操作系统上显示的指针地址一般不是真实的物理内存地址。虚拟操作系统允许程序分布在机器的物理地址空间上。应用程序分为页（或帧），这些页表示内存中的区域。应用程序的页被分配在不同的（可能是不相邻的）内存区域上，而且可能不是同时处于内存中。如果操作系统需要占用被某一页占据的内存，可以将这些内存交换到二级存储器中，待将来需要时再装载进内存中（内存地址一般都会与之前的不同）。这种能力为虚拟操作系统管理内存提供了相当大的灵活性。

每个程序都假定自己能够访问机器的整个物理内存空间，实际上却不是。程序使用的地址是虚拟地址。操作系统会在需要时把虚拟地址映射为物理内存地址。

这意味着页中的代码和数据在程序运行时可能位于不同的物理位置。应用程序的虚拟地址不会变，就是我们在查看指针内容时看到的地址。操作系统会帮我们将虚拟地址映射为真实地址。

操作系统处理一切事务，程序员无法控制也不需要关心。理解这些问题就能解释在虚拟操作系统中运行的程序所返回的地址。

1.1.6 用间接引用操作符解引指针

间接引用操作符（*）返回指针变量指向的值，一般称为解引指针。下面的例子声明和初始化了 num 和 pi：

```
int num = 5;
int *pi = &num;
```

然后下面的语句就用间接引用操作符来显示 5，也就是 num 的值：

```
printf("%p\n",*pi); // 显示5
```

我们也可以把解引操作符的结果用做*左值*。术语“左值”是指赋值操作符左边的操作数，所有的左值都必须可以修改，因为它们会被赋值。

下面的代码把 200 赋给 pi 指向的整数。因为它指向 num 变量，200 会被赋值给 num。图 1-5 说明了这个操作如何影响内存。

```
*pi = 200;
printf("%d\n", num);  // 显示200
```

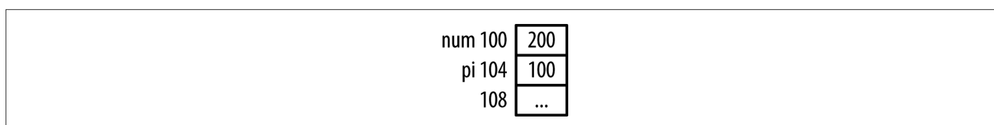


图 1-5：利用解引操作符给内存赋值

1.1.7 指向函数的指针

指针可以声明为指向函数，声明的写法有点难记。下面的代码说明如何声明一个指向函数的指针。函数没有参数也没有返回值。指针的名字是 foo：

```
void (*foo)();
```

指向函数的指针有很多值得讨论的地方，详见第 3 章。

1.1.8 null的概念

null 很有趣，但有时候会被误解。之所以会造成迷惑，是因为我们会遇到几种类似但又不一样的概念，包括：

- null 概念；
- null 指针常量；
- NULL 宏；

- ASCII 字符 NUL；
- null 字符串；
- null 语句。

NULL 被赋值给指针就意味着指针不指向任何东西。null 概念是指指针包含了一个特殊的值，和别的指针不一样，它没有指向任何内存区域。两个 null 指针总是相等的。尽管不常见，但每一种指针类型（如字符指针和整数指针）都可以有对应的 null 指针类型。

null 概念是通过 null 指针常量来支持的一种抽象。这个常量可能是也可能不是常量 0，C 程序员不需要关心实际的内部表示。

NULL 宏是强制类型转换为 void 指针的整数常量 0。在很多库中定义如下：

```
#define NULL    ((void *)0)
```

这就是我们通常理解为 null 指针的东西。这个定义一般可以在多种头文件中找到，包括 stddef.h、stdlib.h 和 stdio.h。

如果编译器用一个非零的位串来表示 null，那么编译器就有责任在指针上下文中把 NULL 或 0 当做 null 指针，实际的 null 内部表示由实现定义。使用 NULL 或 0 是在语言层面表示 null 指针的符号。

ASCII 字符 NUL 定义为全 0 的字节。然而，这跟 null 指针不一样。C 的字符串表示为以 0 值结尾的字符序列。null 字符串是空字符串，不包含任何字符。最后，null 语句就是只有一个分号的语句。

接下来我们会看到，null 指针对于很多数据结构的实现来说都是很有用的特性，比如链表经常用 null 指针来表示链表结尾。

如果要把 null 值赋给 pi，就像下面那样用 NULL：

```
pi = NULL;
```



null 指针和未初始化的指针不同。未初始化的指针可能包含任何值，而包含 NULL 的指针则不会引用内存中的任何地址。

有趣的是，我们可以给指针赋 0，但是不能赋任何别的整数值。看一下下面的赋值操作：

```
pi = 0;
pi = NULL;
```

```
pi = 100; // 语法错误
pi = num; // 语法错误
```

指针可以作为逻辑表达式的唯一操作数。比如说，我们可以用下面的代码来测试指针是否设置成了 NULL。

```
if(pi) {
   // 不是 NULL
} else {
   // 是 NULL
}
```



下面两个表达式都有效，但是有冗余。这样可能更清晰，但是没必要显式地跟 NULL 做比较。

如果这里 pi 被赋了 NULL 值，那就会被解释为二进制 0。在 C 中这表示假，那么倘若 pi 包含 NULL 的话，else 分支就会执行。

```
if(pi == NULL) ...
if(pi != NULL) ...
```



任何时候都不应该对 null 指针进行解引，因为它并不包含合法地址。执行这样的代码会导致程序终止。

1. 用不用NULL

使用指针时哪一种形式更好，NULL 还是 0？无论哪一种都完全没问题，选择哪种只是个人喜好。有些开发者喜欢用 NULL，因为这样会提醒自己是在用指针。另一些人则觉得没必要，因为 NULL 其实就是 0。

然而，NULL 不应该用在指针之外的上下文中。有时候可能有用，但不应该这么用。如果代替 ASCII 字符 NUL 的话肯定会有问题。这个字符没有定义在标准的 C 头文件中。它等于字符 '\0'，其值等于十进制 0。

0 的含义随着上下文的变化而变化，有时候可能是整数 0，有时候又可能是 null 指针。看一下这个例子：

```
int num;
int *pi = 0;    // 这里的 0 表示 null 的指针 NULL
pi = &num;
*pi = 0;        // 这里的 0 表示整数 0
```

我们习惯了重载的操作符，比如星号可以用来声明指针、解引指针或者做乘法。0

也被重载了。我们可能觉得不舒服，因为还没习惯重载操作数。

2. void指针

void 指针是通用指针，用来存放任何数据类型的引用。下面的例子就是一个 void 指针：

```
void *pv;
```

它有两个有趣的性质：

- void 指针具有与 char 指针相同的形式和内存对齐方式；
- void 指针和别的指针永远不会相等，不过，两个赋值为 NULL 的 void 指针是相等的。

任何指针都可以被赋给 void 指针，它可以被转换回原来的指针类型，这样的话指针的值和原指针的值是相等的。在下面的代码中，int 指针被赋给 void 指针然后又被赋给 int 指针：

```
int num;
int *pi = &num;
printf("Value of pi: %p\n", pi);
void* pv = pi;
pi = (int*) pv;
printf("Value of pi: %p\n", pi);
```

运行这段代码后，指针地址是一样的：

```
Value of pi: 100
Value of pi: 100
```

void 指针只用做数据指针，而不能用做函数指针。在 8.4.2 节中，我们将再次研究如何用 void 指针来解决多态的问题。



用 void 指针的时候要小心。如果把任意指针转换为 void 指针，那就没有什么能阻止你再把它转换成不同的指针类型了。

sizeof 操作符可以用在 void 指针上，不过我们无法把这个操作符用在 void 上，如下所示：

```
size_t size = sizeof(void*); // 合法
size_t size = sizeof(void); // 不合法
```

size_t 是用来表示长度的数据类型，会在 1.2.2 节中讨论。

3. 全局和静态指针

指针被声明为全局或静态，就会在程序启动时被初始化为NULL。下面是全局和静态指针的例子：

```
int *globalpi;

void foo() {
   static int *staticpi;
   ...
}

int main() {
   ...
}
```

图 1-6 说明了内存布局，栈帧被推入栈中，堆用来动态分配内存，堆上面的区域用来存放全局 / 静态变量。这只是原理图，静态和全局变量一般放在与栈和堆所处的数据段不同的数据段中。栈和堆将在 3.1 节讨论。

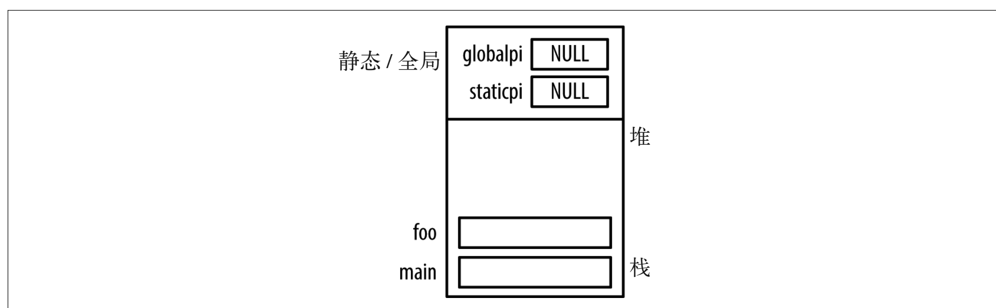


图 1-6：全局和静态指针的内存分配

1.2 指针的长度和类型

如果考虑应用程序的兼容性和可移植性，指针长度就是一个问题。在大部分现代平台上，数据指针的长度通常是一样的，与指针类型无关，char指针和结构体指针长度相同。尽管 C 标准没有规定所有数据类型的指针长度相同，但是通常实际情况就是这样。不过，函数指针长度可能与数据指针长度不同。

指针长度取决于使用的机器和编译器。比如，在现代 Windows 上，指针是 32 位或 64 位长。对于 DOS 和 Windows 3.1 来说，指针则是 16 位或 32 位长。

1.2.1 内存模型

64 位机器的出现导致为不同数据类型分配的内存在长度上的差异变得明显。不同的

机器和编译器在给 C 的基本数据类型分配空间上有不同的做法。用来描述不同数据模型的一种通用表示法总结如下：

```
I In L Ln LL LLn P Pn
```

每个大写字母对应整数、长整数或指针，小写字母表示为该数据类型分配的位数。表 1-3 总结了这些模型，其中数字表示位数。

表1-3：机器内存模型

C数据类型	LP64	ILP64	LLP64	ILP32	LP32
`char`	8	8	8	8	8
`short`	16	16	16	16	16
`_int32`		32			
`int`	32	64	32	32	16
`long`	64	64	32	32	32
`long long`			64		
`pointer`	64	64	64	32	32

模型取决于操作系统和编译器，一种操作系统可能支持多种模型，这通常是用编译器选项来控制的。

1.2.2 指针相关的预定义类型

使用指针时经常用到以下四种预定义类型。

- `size_t`
  用于安全地表示长度。

- `ptrdiff_t`
  用于处理指针算术运算。

- `intptr_t`和`uintptr_t`
  用于存储指针地址。

下面将展示每种类型的用法，`ptrdiff_t` 除外，我们会在 1.3.1 节的“两个指针相减”中讨论它。

1. 理解`size_t`

`size_t` 类型表示 C 中任何对象所能达到的最大长度。它是无符号整数，因为负数在这里没有意义。它的目的是提供一种可移植的方法来声明与系统中可寻址的内存区域一致的长度。`size_t` 用做 `sizeof` 操作符的返回值类型，同时也是很多函数的

参数类型，包括 malloc 和 strlen。

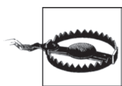


在声明诸如字符数或者数组索引这样的长度变量时用 size_t 是好的做法。它经常用于循环计数器、数组索引，有时候还用在指针算术运算上。

size_t 的声明是实现相关的。它出现在一个或多个标准头文件中，比如 stdio.h 和 stblib.h，典型的定义如下：

```
#ifndef __SIZE_T
#define __SIZE_T
typedef unsigned int size_t;
#endif
```

define 指令确保它只被定义一次。实际的长度取决于实现。通常在 32 位系统上它的长度是 32 位，而在 64 位系统上则是 64 位。一般来说，size_t 可能的最大值是 SIZE_MAX。



通常 size_t 可以用来存放指针，但是假定 size_t 和指针一样长不是个好主意。稍后的“使用 sizeof 操作符和指针”会讲到，intptr_t 是更好的选择。

打印 size_t 类型的值时要小心。这是无符号值，如果选错格式说明符，可能会得到不可靠的结果。推荐的格式说明符是 %zu。不过，某些情况下不能用这个说明符，作为替代，可以考虑 %u 或 %lu。

下面这个例子将一个变量定义为 size_t，然后用两种不同的格式说明符来打印：

```
size_t sizet = -5;
printf("%d\n",sizet);
printf("%zu\n",sizet);
```

因为 size_t 本来是用于表示正整数的，如果用来表示负数就会出问题。如果为其赋一个负数，然后用 %d 和 %zu 格式说明符打印，就得到如下结果：

```
-5
4294967291
```

%d 把 size_t 当做有符号整数，它打印出 −5 因为变量中存放的就是 −5。%zu 把 size_t 当做无符号整数。当 −5 被解析为有符号数时，高位置为 1，表示这个数是负数。当它被解析为无符号数时，高位的 1 被当做 2 的乘幂。所以在用 %zu 格式说明符时才会看到那个大整数。

正数会正常显示，如下所示：

```
sizet = 5;
printf("%d\n",sizet); // 显示5
printf("%zu\n",sizet); // 显示5
```

因为 size_t 是无符号的，一定要给这种类型的变量赋正数。

2. 对指针使用sizeof操作符

sizeof 操作符可以用来判断指针长度。下面的代码显示 char 指针的长度：

```
printf("Size of *char: %d\n",sizeof(char*));
```

输出如下：

```
Size of *char: 4
```



当需要用指针长度时，一定要用 sizeof 操作符。

函数指针的长度是可变的。通常，对于给定的操作系统和编译器组合，它是固定的。很多编译器支持创建 32 位和 64 位应用程序，所以对于同一个程序来说，不同的编译选项可能会导致其使用不同的指针长度。

在 Harvard 架构上，代码和数据存储在不同的物理内存中。比如 Intel 的 MCS-51（8051）微处理器就是 Harvard 架构。尽管 Intel 不再生产这种芯片，但现在还是有很多二进制兼容的衍生品在使用。Small Device C Complier（SDCC）就支持这类处理器（参见 http://sdcc.sourceforge.net/doc/sdccman.pdf）。这种机器的指针长度可能介于 1 到 4 字节之间，因此指针长度应该在需要时再确定，原因是在这种环境中它并不固定。

3. 使用intptr_t和uintptr_t

intptr_t 和 uintptr_t 类型用来存放指针地址。它们提供了一种可移植且安全的方法声明指针，而且和系统中使用的指针长度相同，对于把指针转化成整数形式来说很有用。

uintptr_t 是 intptr_t 的无符号版本。对于大部分操作，用 intptr_t 比较好。uintptr_t 不像 intptr_t 那样灵活。下面的例子说明如何使用 intptr_t：

```
int num;
intptr_t *pi = &num;
```

如果像下面那样试图把整数地址赋给 uintptr_t 类型的指针，我们会得到一个语法错误：

```
uintptr_t *pu = &num;
```

错误看起来是这样的：

```
error: invalid conversion from 'int*' to
       'uintptr_t* {aka unsigned int*}' [-fpermissive]
```

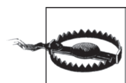
不过，用强制类型转换来赋值是可以的：

```
intptr_t *pi = &num;
uintptr_t *pu = (uintptr_t*)&num;
```

如果不转换类型，不能将 uintptr_t 用于其他类型：

```
char c;
uintptr_t *pc = (uintptr_t*)&c;
```

当可移植性和安全性变得重要时，就应该使用这些类型。不过，为简单起见，我们的例子中不会使用。



避免把指针转换成整数。如果指针是 64 位，整数只有 4 字节时就会丢失信息。



早期的 Intel 处理器采用 16 位的分段架构，近指针和远指针也是相对的。今天的虚拟内存架构上就不是这样了。远指针和近指针是 C 标准的扩展，用来支持早期的 Intel 处理器的分段架构。近指针一次只能寻址 64 KB 的内存。远指针最多可以寻址 1 MB 内存，但是比近指针慢。巨指针是规范化过的远指针，使用尽可能高的段。

1.3 指针操作符

指针有几类操作符。目前我们已经接触过解引和取地址操作符，本节将近距离研究指针算术运算和比较。表 1-4 总结了指针操作符。

表1-4：指针操作符

操 作 符	名 称	含 义
*		用来声明指针
*	解引	用来解引指针
->	指向	用来访问指针引用的结构的字段
+	加	用于对指针做加法
-	减	用于对指针做减法
== !=	相等、不等	比较两个指针
> >= < <=	大于、大于等于、小于、小于等于	比较两个指针
(数据类型)	转换	改变指针的类型

1.3.1 指针算术运算

数据指针可以执行以下几种算术运算：

- 给指针加上整数；
- 从指针减去整数；
- 两个指针相减；
- 比较指针。

函数指针则不一定。

1. 给指针加上整数

这种操作很普遍也很有用。给指针加上一个整数实际上加的数是这个整数和指针数据类型对应字节数的乘积。

各个系统的基本数据类型长度可能不同，正如1.2.1节所述。表1-5显示了大部分系统的常见长度，除非特别指定，本书的示例会使用这里的值。

表1-5：数据类型长度

数据类型	长度（字节）
byte	1
char	1
short	2
int	4
long	8
float	4
double	8

为了说明给指针加上整数的效果，我们会使用一个整数数组，如下所示。每次pi加1，地址就加4。这些变量的内存分配如图1-7所示。指针是用数据类型声明的，

以便执行算术运算。这种自动调整指针值的可移植方法之所以可能，前提就是知道数据类型的大小。

```
int vector[] = {28, 41, 7};
int *pi = vector;       // pi: 100

printf("%d\n",*pi);     // 显示28
pi += 1;                // pi: 104
printf("%d\n",*pi);     // 显示41
pi += 1;                // pi: 108
printf("%d\n",*pi);     // 显示7
```



如果这里使用数组的名字，返回的只是数组地址，也就是数组第一个元素的地址。

vector[0] 100 | 28
vector[1] 104 | 41
vector[2] 108 | 7
pi 112 | 100

图 1-7：vector 数组的内存分配情况

在下面的代码中，我们给指针加 3，pi 变量会包含地址 112，就是 pi 本身的地址：

```
pi = vector;
pi += 3;
```

指针指向了自己，这样没什么用，但是说明了在做指针算术运算时要小心。访问超出数组范围的内存很危险，应该避免。没有什么能保证被访问的内存是有效变量，存取无效或无用地址的情况很容易发生。

下面的代码演示了 short 和 char 类型指针的加法操作：

```
short s;
short *ps = &s;
char c;
char *pc = &c;
```

我们假设内存分配如图 1-8 所示，这里用到的地址以 4 字节为界。真实的地址可能涉及不同的字节数和不同的字节序。

s 120 | ...
ps 124 | 120
c 128 | ...
pc 132 | 128

图 1-8：short 和 char 指针

下面的代码给每个指针加 1 然后显示内容：

```
printf("Content of ps before: %d\n",ps);
ps = ps + 1;
printf("Content of ps after: %d\n",ps);

printf("Content of pc before: %d\n",pc);
pc = pc + 1;
printf("Content of pc after: %d\n",pc);
```

运行后，你应该能得到类似如下的结果：

```
Content of ps before: 120
Content of ps after:  122
Content of pc before: 128
Address of pc after:  129
```

`ps` 指针增加了 2，因为 `short` 的长度是 2 字节。`pc` 指针增加了 1，因为它的数据类型长 1 字节。这些地址可能没有包含有用的信息。

2. void指针和加法

作为扩展，大部分编译器都允许给 `void` 指针做算术运算，这里我们假设 `void` 指针的长度是 4。不过，试图给 `void` 指针加 1 可能导致语法错误。在下面的代码片段中，我们声明指针并试图给它加 1：

```
int num = 5;
void *pv = &num;
printf("%p\n",pv);
pv = pv+1;        //语法警告
```

下面是警告信息：

```
warning: pointer of type 'void *' used in arithmetic [-Wpointerarith]
```

这不是标准 C 允许的行为，所以编译器发出了警告。不过，`pv` 包含的地址增加了 4 字节。

3. 从指针减去整数

就像整数可以和指针相加一样，也能从指针减去整数。减去整数时，地址值会减去数据类型的长度和整数值的乘积。为了演示从指针减去整数的效果，我们使用如下所示的数组。这些变量的内存分配如图 1-7 所示。

```
int vector[] = {28, 41, 7};
int *pi = vector + 2;   // pi: 108

printf("%d\n",*pi);     // 显示 7
```

```
pi--;                    // pi: 104
printf("%d\n",*pi);      // 显示 41
pi--;                    // pi: 100
printf("%d\n",*pi);      // 显示 28
```

pi 每次减 1，地址都会减 4。

4. 指针相减

一个指针减去另一个指针会得到两个地址的差值。这个差值通常没什么用，但可以判断数组中的元素顺序。

指针之间的差值是它们之间相差的“单位”数，差的符号取决于操作数的顺序。这和指针加法是一样的，加到指针上的是数据的长度。我们把“单位”当做操作数。在下例中，我们声明一个数组和数组元素的指针，然后相减：

```
int vector[] = {28, 41, 7};
int *p0 = vector;
int *p1 = vector+1;
int *p2 = vector+2;

printf("p2-p0: %d\n",p2-p0); // p2-p0: 2
printf("p2-p1: %d\n",p2-p1); // p2-p1: 1
printf("p0-p1: %d\n",p0-p1); // p0-p1: -1
```

在第一个 printf 语句中，我们看到数组的最后一个和第一个元素的位置相差 2，就是说它们的索引值相差 2。在最后一个 printf 语句中，差值是 −1，表示 p0 在 p1 前面，而且它们紧挨着。图 1-9 说明了本例中内存的分配情况。

vector[0] 100	28
vector[1] 104	41
vector[2] 108	7
p0 112	100
p1 116	104
p2 120	108

图 1-9：指针相减

ptrdiff_t 类型表示两个指针差值的可移植方式。在上例中，指针相减的结果以 ptrdiff_t 类型返回。因为指针长度可能不同，这个类型简化了处理差值的任务。

不要把这种技术和利用解引操作来做数字相减混淆。在下例中，我们用指针来确定数字中第一个元素和第二个元素中存储的值的差。

```
printf("*p0-*p1: %d\n",*p0-*p1); // *p0-*p1: -13
```

1.3.2 比较指针

指针可以用标准的比较操作符来比较。通常，比较指针没什么用。然而，当把指针和数组元素相比时，比较结果可以用来判断数组元素的相对顺序。

我们仍然用前面“指针相减”中使用的 `vector` 数组来说明指针的比较。这里用到了几种比较操作符，结果为 1 表示真，为 0 表示假：

```
int vector[] = {28, 41, 7};
int *p0 = vector;
int *p1 = vector+1;
int *p2 = vector+2;

printf("p2>p0: %d\n",p2>p0);     // p2>p0: 1
printf("p2<p0: %d\n",p2<p0);     // p2<p0: 0
printf("p0>p1: %d\n",p0>p1);     // p0>p1: 0
```

1.4 指针的常见用法

指针用处很多。在本节中，我们探讨指针的不同用法，包括：

- 多层间接引用；
- 常量指针。

1.4.1 多层间接引用

指针可以用不同的间接引用层级。把变量声明为指针的指针并不少见，有时候称它们为*双重指针*。一个很好的例子就是用传统的 `argv` 和 `argc` 参数来给 `main` 函数传递程序参数，第 5 章将详细讨论。

下例使用了三个数组。第一个数组是用来存储书名列表的字符串数组：

```
char *titles[] = {"A Tale of Two Cities",
       "Wuthering Heights","Don Quixote",
       "Odyssey","Moby-Dick","Hamlet",
       "Gulliver's Travels"};
```

还有两个数组分别用来维护一个“畅销书”列表和一个英文书列表。这两个数组保存的是 `titles` 数组里书名的地址，而不是书名的副本。两个数组都声明为字符指针的指针。数组元素会保存 `titles` 数组中元素的地址，这样可以避免对每个书名重复分配内存，确保每个书名的位置唯一。如果需要修改书名，只改一个地方就可以了。

另外两个数组声明如下。每个数组元素包含一个指向 `char` 指针的指针。

```
char **bestBooks[3];
char **englishBooks[4];
```

接下来初始化这两个数组，然后打印其中一个元素，如下所示。在赋值语句中，右边的值是通过先做下标索引再取地址的操作得到的。比如说第二个语句把 `titles` 数组中第 4 个元素的地址赋给 `bestBooks` 的第 2 个元素：

```
bestBooks[0] = &titles[0];
bestBooks[1] = &titles[3];
bestBooks[2] = &titles[5];

englishBooks[0] = &titles[0];
englishBooks[1] = &titles[1];
englishBooks[2] = &titles[5];
englishBooks[3] = &titles[6];

printf("%s\n",*englishBooks[1]);   // Wuthering Heights
```

本例的内存分配如图 1-10 所示。

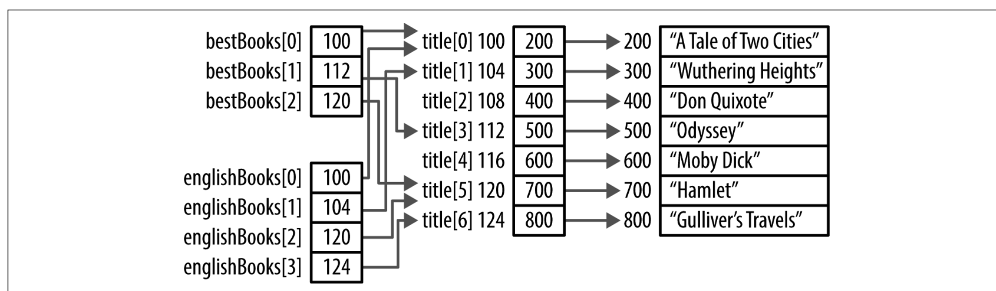


图 1-10：指针的指针

用多层间接引用可以为代码的编写和使用提供更多的灵活性，否则有些操作实现起来会困难一些。在本例中，如果书名的地址变了，那么只需要修改 `titles` 数组即可，不需要修改其他两个数组。

间接引用没有层数限制，当然，使用的层数过多会让人迷惑，很难维护。

1.4.2　常量与指针

C 语言的功能强大而丰富，还表现在 `const` 关键字与指针的结合使用上。对不同的问题，它能提供不同的保护。特别强大和有用的是指向常量的指针。在第 3 章和第 5 章，我们将看到如何用这种技术来阻止函数的使用者修改函数的参数。

1. 指向常量的指针

可以将指针定义为指向常量，这意味着不能通过指针修改它所引用的值。下例声明

了一个整数和一个整数常量，然后声明了一个整数指针和一个指向整数常量的指针，并分别初始化为对应的整数：

```
int num = 5;
const int limit = 500;
int *pi;                  // 指向整数
const int *pci;           // 指向整数常量

pi = &num;
pci = &limit;
```

图 1-11 是它们的内存分配情况。

图 1-11：指向整数常量的指针

下面的代码会打印这些变量的地址和值：

```
printf("  num - Address: %p  value: %d\n",&num, num);
printf("limit - Address: %p  value: %d\n",&limit, limit);
printf("   pi - Address: %p  value: %p\n",&pi, pi);
printf("  pci - Address: %p  value: %p\n",&pci, pci);
```

运行代码会产生类似下面的输出：

```
  num - Address: 100  value: 5
limit - Address: 104  value: 500
   pi - Address: 108  value: 100
  pci - Address: 112  value: 104
```

如果只是读取整数的值，那么引用指向常量的指针就没事，读取是完全合法的，而且也是必要的功能，如下所示：

```
printf("%d\n", *pci);
```

我们不能解引指向常量的指针并改变指针所引用的值，但可以改变指针。指针的值不是常量。指针可以改为引用另一个整数常量，或者普通整数。这样做不会有问题。声明只是限制我们不能通过指针来修改引用的值。

这意味着下面的赋值是合法的：

```
pci = &num;
```

我们可以解引 `pci` 来读取它，但不能解引它来修改它。

考虑下面的赋值语句：

```
*pci = 200;
```

这会导致如下语法错误：

```
'pci' : you cannot assign to a variable that is const
```

指针认为自己指向的是整数常量，所以不允许用指针来修改这个整数。我们还是可以通过名字来修改 num 变量，只是不能用 pci 来修改。

理论上来说，常量的指针也可以如图 1-12 那样可视化，普通方框表示变量，阴影方框表示常量。pci 指向的阴影方框不能用 pci 来修改，虚线表示指针可以引用的数据类型。在上例中，pci 指向 limit。

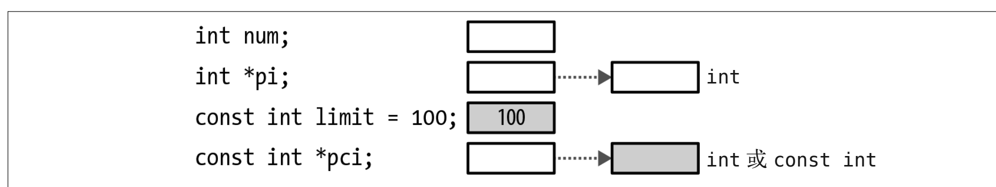


图 1-12：指向常量的指针

把 pci 声明为指向整数常量的指针意味着：

- pci 可以被修改为指向不同的整数常量；
- pci 可以被修改为指向不同的非整数常量；
- 可以解引 pci 以读取数据；
- 不能解引 pci 从而修改它指向的数据。



数据类型和 const 关键字的顺序不重要。下面两个语句是等价的：

```
const int *pci;
int const *pci;
```

2. 指向非常量的常量指针

也可以声明一个指向非常量的常量指针。这么做意味着指针不可变，但是它指向的数据可变。下面是这种指针的例子；

```
int num;
int *const cpi = &num;
```

有了这个声明：

- cpi 必须被初始化为指向非常量变量；
- cpi 不能被修改；
- cpi 指向的数据可以被修改。

从原理上说，这类指针可以用图 1-13 来说明。

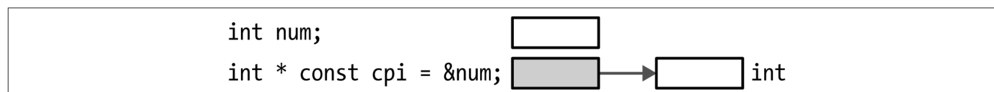


图 1-13：指向非常量的常量指针

无论 cpi 引用了什么，都可以解引 cpi 然后赋一个新值。下面是两条合法的赋值语句：

```
*cpi = limit;
*cpi = 25;
```

然而，如果我们试图把 cpi 初始化为指向常量 limit，如下所示：

```
const int limit = 500;
int *const cpi = &limit;
```

那么就会产生一个警告：

```
warning: initialization discards qualifiers from pointer target type
```

如果这里 cpi 引用了常量 limit，那常量就可以修改了。这样不对，因为常量是不能被修改的。

在把地址赋给 cpi 之后，就不能像下面这样再赋给它一个新值了：

```
int num;
int age;
int *const cpi = &num;
cpi = &age;
```

如果采用这种做法会产生如下错误信息：

```
'cpi' : you cannot assign to a variable that is const
```

3. 指向常量的常量指针

指向常量的常量指针很少派上用场。这种指针本身不能修改，它指向的数据也不能通过它来修改。下面是指向常量的常量指针的一个例子：

```
const int * const cpci = &limit;
```

指向常量的常量指针可以用图 1-14 来说明。

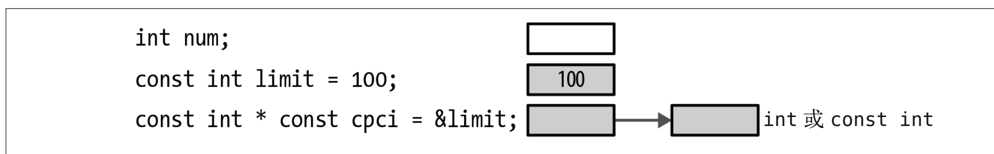


图 1-14：指向常量的常量指针

与指向常量的指针类似，不一定只能将常量的地址赋给 cpci。如下所示，我们其实还可以把 num 的地址赋给 cpci：

```
int num;
const int * const cpci = &num;
```

声明指针时必须进行初始化。如果像下面这样不进行初始化：

```
const int * const cpci;
```

就会产生如下语法错误：

```
'cpci' : const object must be initialized if not extern
```

对于指向常量的常量指针，我们不能：

- 修改指针；
- 修改指针指向的数据。

重新赋给 cpci 一个新地址：

```
cpci = &num;
```

会导致如下语法错误：

```
'cpci' : you cannot assign to a variable that is const
```

像下面这样试图解引指针并赋新值：

```
*cpci = 25;
```

会产生如下错误：

```
'cpci' : you cannot assign to a variable that is const
expression must be a modifiable lvalue
```

不过，指向常量的常量指针很少用到。

4. 指向“指向常量的常量指针”的指针

指向常量的指针也可以有多层间接引用。在下例中，我们声明一个指向上一节提到

的 cpci 指针的指针。从右往左读可以帮助我们理解这个声明：

```
const int * const cpci = &limit;
const int * const * pcpci;
```

指向“指向常量的常量指针”的指针可以用图 1-15 来说明。

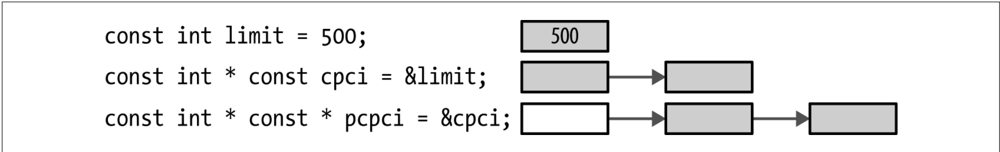


图 1-15：指向“指向常量的常量指针”的指针

下面说明它们的使用。这段代码的输出应该是两个 500：

```
printf("%d\n",*cpci);
pcpci = &cpci;
printf("%d\n",**pcpci);
```

下表总结了本节讨论的前四种指针。

指针类型	指针是否可修改	指向指针的数据是否可修改
指向非常量的指针	是	是
指向常量的指针	是	否
指向非常量的常量指针	否	是
指向常量的常量指针	否	否

1.5　小结

本章讨论了指针的基本概念，包括如何声明指针，在常见的场景中如何使用指针。我们也提到了 null 的有趣概念和它的变种，还有一系列指针操作符。

我们知道了指针的长度是可变的，它取决于目标系统和编译器支持的内存模型。我们也探索了 const 关键字和指针一起使用的问题。

有了这些基础知识，下一步就可以探讨那些指针可以大显身手的领域了。这些领域包括把指针作为函数参数、辅助创建数据结构，以及指针在动态内存分配中的应用。另外，我们也会看到指针如何让应用程序更安全。

第2章

C的动态内存管理

指针的强大很大程度上源于它们能追踪动态分配的内存。通过指针来管理这部分内存是很多操作的基础，包括一些用来处理复杂数据结构的操作。要完全利用这些能力，需要理解C的动态内存管理是怎么回事。

C程序在运行时环境中执行，这通常是由操作系统提供的环境，支持栈和堆以及其他的程序行为。

内存管理对所有程序来说都很重要。有时候内存由运行时系统隐式地管理，比如为自动变量分配内存。在这种情况下，变量分配在它所处的函数的栈帧上。如果是静态或全局变量，内存处于程序的数据段，会被自动清零。数据段是一个区别于可执行代码和运行时系统管理的其他数据的内存区域。

由于可以先分配内存然后释放，因而应用程序可以更灵活高效地管理内存，无需为适应数据结构可能的最大长度分配内存，只要分配实际需要的内存即可。

比如，在C99以前数组是固定长度的。如果我们要持有可变数量的元素，比如员工记录，可能就必须声明一个足够大的数组来装下可能的最大员工数。如果我们低估了这个值，那就只能重新编译应用程序或是采用别的办法。如果高估了，那就会浪费空间。动态分配内存的能力也对使用链表或队列等可变数量元素的数据结构有帮助。



C99引入了变长数组（VLA）。数组长度在运行时而不是编译时确定。不过，数组一旦创建出来就不能再改变长度了。

像 C 这类语言也支持动态内存管理，对象就是从堆上分配出来的内存。这是用分配和释放函数手动实现的，这个过程被称为*动态内存管理*。

本章一开始我们概述如何分配和释放内存。接下来讲解基本的分配函数，如 `malloc` 和 `realloc`，还会讨论 `free` 函数，包括 `NULL` 的使用和重复释放这类问题。

迷途指针是个常见问题。我们会通过示例说明什么情况下出现迷途指针以及处理这种问题的技术。最后一节讲解内存管理的其他技术。指针使用不当会造成不可预期的行为，这么说的意思是程序可能产生无效结果，损坏数据或者终止程序。

2.1　动态内存分配

在 C 中动态分配内存的基本步骤有：

(1) 用 `malloc` 类的函数分配内存；
(2) 用这些内存支持应用程序；
(3) 用 `free` 函数释放内存。

这个方法在具体操作上可能存在一些小变化，不过这里列出的是最常见的。在下例中，我们用 `malloc` 函数为整数分配内存。指针将分配的内存赋值为 5，然后内存被 `free` 函数释放。

```
int *pi = (int*) malloc(sizeof(int));
*pi = 5;
printf("*pi: %d\n", *pi);
free(pi);
```

当这段代码执行时会打印数字 5。图 2-1 说明了在 `free` 函数执行之前内存如何分配。为方便在本章说明问题，除非特别指出，我们假定示例代码出现在 `main` 函数中。

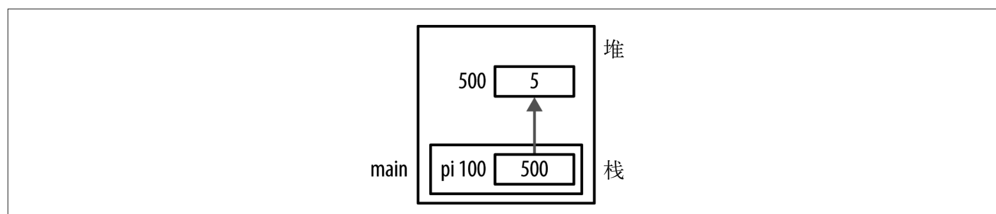


图 2-1：整数的内存分配

`malloc` 函数的参数指定要分配的字节数。如果成功，它会返回从堆上分配的内存的指针。如果失败则会返回空指针。测试所分配内存的指针是否有效在 2.2.1 节中讨论。`sizeof` 操作符使应用程序更容易移植，还能确定在宿主系统中应该分配的

正确的字节数。

在本例中，我们试图为整数分配足够多的内存。假定长度是 4，我们可以这么写：

```
int *pi = (int*) malloc(4);
```

然而，依赖于系统所用的内存模型，整数的长度可能会发生变化。可移植的方法是使用 sizeof 操作符，这样不管程序在哪里运行都会返回正确的长度。



涉及解引操作的常见错误见下面的代码：

```
int *pi;
*pi = (int*) malloc(sizeof(int));
```

问题出在赋值符号的左边。我们在解引指针，这样会把 malloc 函数返回的地址赋给 pi 中存放的地址所在的内存单元。如果这是第一次对指针进行赋值操作，那指针所包含的地址可能无效。正确的方法如下所示：

```
pi = (int*) malloc(sizeof(int));
```

这种情况下不应该用解引操作符。

稍后也会深入讨论 free 函数，它和 malloc 协同工作，不再需要内存时将其释放。



每次调用 malloc（或类似函数），程序结束时必须有对应的 free 函数调用，以防止内存泄漏。

一旦内存被释放，就不应该再访问它了。通常我们不会在释放内存后有意去访问，不过，就像 2.4 节中所说的，这也有可能意外发生。在这种情况下系统的行为将依赖于实现。通常的做法是总是把被释放的指针赋值为 NULL，2.3.1 节会讨论这一点。

分配内存时，堆管理器维护的数据结构中会保存额外的信息。这些信息包括块大小和其他一些东西，通常放在紧挨着分配块的位置。如果应用程序的写入操作超出了这块内存，数据结构可能会被破坏。这可能会造成程序奇怪的行为或者堆损坏，第 7 章会演示相关示例。

考虑如下代码段，我们为字符串分配内存，让它可以存放最多 5 个字符外加结尾的 NUL 字符。for 循环在每个位置写入 0，但是没有在写入 6 字节后停止。for 语句的结束条件是写入 8 字节。写入的 0 是二进制 0 而不是 ASCII 字符 0 的值。

```
char *pc = (char*) malloc(6);
for(int i=0; i<8; i++) {
    pc[i] = 0;
}
```

在图 2-2 中，6 字节的字符串后面还分配了额外的内存，这是堆管理器用来记录内存分配的。如果我们越过字符串的结尾边界写入，额外的内存中的数据会损坏。在本例中，额外的内存跟在字符串后面。不过，实际的位置和原始信息取决于编译器。

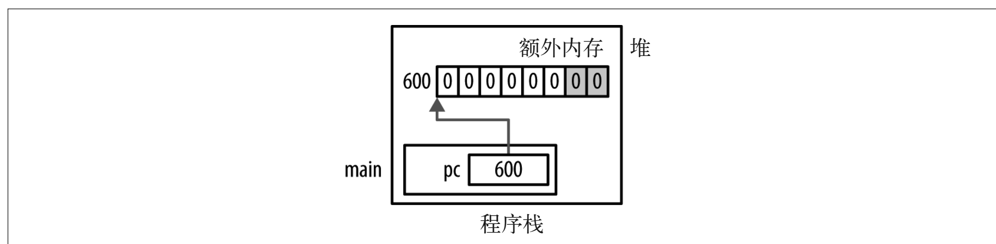


图 2-2：堆管理器用到的额外内存

内存泄漏

如果不再使用已分配的内存却没有将其释放就会发生内存泄漏，导致内存泄漏的情况可能如下：

- 丢失内存地址；
- 应该调用 free 函数却没有调用（有时候也称为隐式泄漏）。

内存泄漏的一个问题是无法回收内存并重复利用，堆管理器可用的内存将变少。如果内存不断地被分配并丢失，那么当需要更多内存而 malloc 又不能分配时程序可能会终止，因为它用光了内存。在极端情况下，操作系统可能崩溃。

下面这个简单的例子可以说明这个问题：

```
char *chunk;
while (1) {
    chunk = (char*) malloc(1000000);
    printf("Allocating\n");
}
```

chunk 变量指向堆上的内存。然而，在它指向另一块内存之前没有释放这块内存。最终，程序会用光内存然后非正常终止，即使没有终止，至少内存的利用效率也不高。

1. 丢失地址

下面的代码段说明了当 pi 被赋值为一个新地址时丢失内存地址的例子。当 pi 又指向第二次分配的内存时，第一次分配的内存的地址就会丢失。

```
int *pi = (int*) malloc(sizeof(int));
*pi = 5;
...
pi = (int*) malloc(sizeof(int));
```

图 2-3 说明了这一点，“前”和“后”分别表示在执行第二次 malloc 之前和之后的程序状态。由于没有释放地址 500 处的内存，程序已经没有地方持有这个地址。

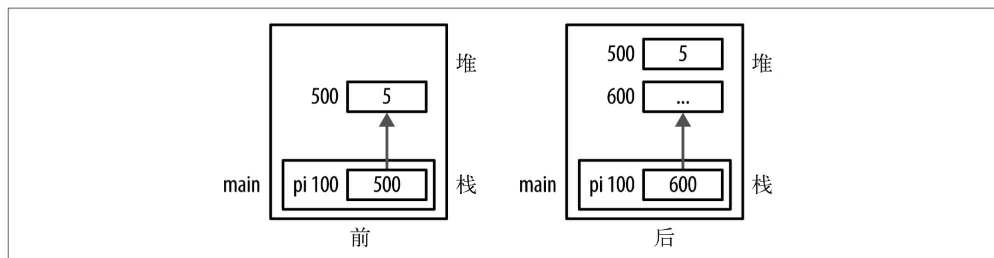


图 2-3：丢失地址

下面这个例子是为字符串分配内存，将其初始化，并逐个字符打印字符串：

```
char *name = (char*)malloc(strlen("Susan")+1);
strcpy(name,"Susan");
while(*name != 0) {
    printf("%c",*name);
    name++;
}
```

然而每次迭代 name 都会增加 1，最后 name 会指向字符串结尾的 NUL 字符，如图 2-4 所示，分配内存的起始地址丢失了。

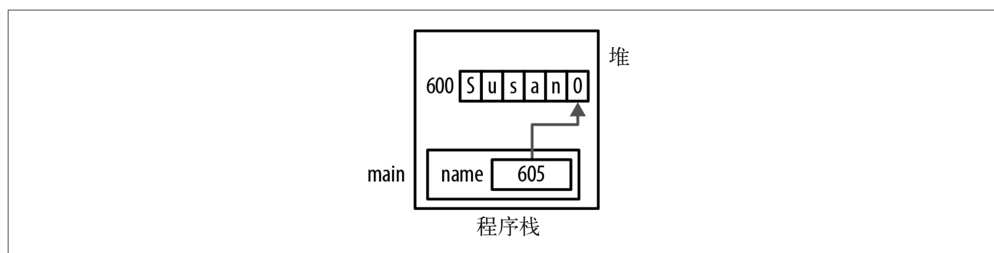


图 2-4：丢失动态分配的内存的地址

2. 隐式内存泄漏

如果程序应该释放内存而实际却没有释放，也会发生内存泄漏。如果我们不再需要某个对象但它仍然保存在堆上，就会发生隐式内存泄漏，一般这是程序员忽视所致。这类泄漏的主要问题是对象在使用的内存其实已经不需要了，应该归还给堆。最差

的情况是，堆管理器可能无法按需分配内存，导致程序不得不终止。最好的情况是我们持有了不必要的内存。

在释放用 `struct` 关键字创建的结构体时也可能发生内存泄漏。如果结构体包含指向动态分配的内存的指针，那么可能需要在释放结构体之前先释放这些指针，第 6 章会展示示例。

2.2 动态内存分配函数

有几个内存分配函数可以用来管理动态内存，虽然具体可用的函数取决于系统，但大部分系统的 stdlib.h 头文件中都有如下函数：

- `malloc`
- `realloc`
- `calloc`
- `free`

表 2-1 总结了这些函数。

表2-1：动态内存分配函数

函　数	描　述
`malloc`	从堆上分配内存
`realloc`	在之前分配的内存块的基础上，将内存重新分配为更大或者更小的部分
`calloc`	从堆上分配内存并清零
`free`	将内存块返回堆

动态内存从堆上分配，至于一连串内存分配调用，系统不保证内存的顺序和所分配内存的连续性。不过，分配的内存会根据指针的数据类型对齐，比如说，4 字节的整数会分配在能被 4 整除的地址边界上。堆管理器返回的地址是最低字节的地址。

在图 2-3 中，`malloc` 函数在地址 500 处分配了 4 字节空间，第二次使用该函数在地址 600 处分配了内存。它们都处于 4 字节地址边界上，而且不是从相邻的内存位置上分配的。

2.2.1 使用malloc函数

`malloc` 函数从堆上分配一块内存，所分配的字节数由该函数唯一的参数指定，返回值是 `void` 指针，如果内存不足，就会返回 `NULL`。此函数不会清空或者修改内存，所以我们认为新分配的内存包含垃圾数据。函数的原型如下：

```
void* malloc(size_t);
```

这个函数只有一个参数，类型是 size_t，我们在第 1 章讨论过此类型。传递参数给这个函数时要小心，因为如果参数是负数就会引发问题。在有些系统中，参数是负数会返回 NULL。

如果 malloc 的参数是 0，其行为是实现相关的：可能返回 NULL 指针，也可能返回一个指向分配了 0 字节区域的指针。如果 malloc 函数的参数是 NULL，那么一般会生成一个警告然后返回 0 字节。

以下是 malloc 函数的典型用法：

```
int *pi = (int*) malloc(sizeof(int));
```

执行 malloc 函数时会进行以下操作：

(1) 从堆上分配内存；
(2) 内存不会被修改或是清空；
(3) 返回首字节的地址。



因为当 malloc 无法分配内存时会返回 NULL，在使用它返回的指针之前先检查 NULL 是不错的做法，如下所示：

```
int *pi = (int*) malloc(sizeof(int));
if(pi != NULL) {
   // 指针没有问题
} else {
   // 无效的指针
}
```

1. 要不要强制类型转换

C 引入 void 指针之前，在两种互不兼容的指针类型之间赋值需要对 malloc 使用显式转换类型以避免产生警告。因为可以将 void 指针赋值给其他任何指针类型，所以就不再需要显式类型转换了。有些开发者认为显式类型转换是不错的做法，因为：

- 这样可以说明 malloc 函数的用意；
- 代码可以和 C++（或早期的 C 编译器）兼容，后两者需要显式的类型转换。

如果没有引用 malloc 的头文件，类型转换可能会有问题，编译器可能会产生警告。C 默认函数返回整数，如果没有引用 malloc 的原型，编译器会抱怨你试图把 int 赋值给指针。

2. 分配内存失败

如果声明了一个指针，但没有在使用之前为它指向的地址分配内存，那么内存通常会包含垃圾，这往往会导致一个无效内存引用的错误。考虑如下代码片段：

```
int *pi;
...
printf("%d\n",*pi);
```

内存分配如图 2-5 所示。这个问题会在第 7 章详细讨论。

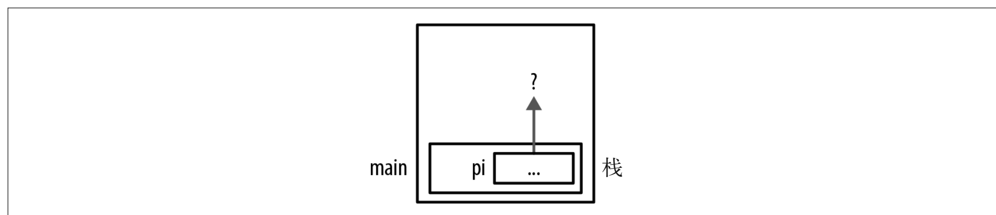


图 2-5：没有分配内存

执行这段代码可能会导致一个运行时异常。字符串中这类问题比较常见，如下所示：

```
char *name;
printf("Enter a name: ");
scanf("%s",name);
```

这里使用的是 name 所引用的内存，看起来似乎可以正确执行，实际上这块内存还没有分配。把图 2-5 中的 pi 换成 name 就可以说明这个问题。

3. 没有给malloc传递正确的参数

malloc 函数分配的字节数是由它的参数指定的，在用这个函数分配正确的字节数时要小心。比如说要为 10 个双精度浮点数分配空间，那就需要 80 字节，通过下面的代码可以做到：

```
double *pd = (double*)malloc(NUMBER_OF_DOUBLES * sizeof(double));
```



为数据类型分配指定字节数时尽量用 sizeof 操作符。

下例尝试为 10 个双精度浮点数分配内存：

```
const int NUMBER_OF_DOUBLES = 10;
double *pd = (double*)malloc(NUMBER_OF_DOUBLES);
```

这段代码实际只分配了 10 字节。

4. 确认所分配的内存数

没有标准的方法可以知道堆上分配的内存总数，不过有些编译器为此提供了扩展。另外，也没有标准的方法可以知道堆管理器分配的内存块大小。

比如说，如果我们为一个字符串分配 64 字节，堆管理器会分配额外的内存来管理这个块。所分配内存的总大小，以及堆管理器所用到的内存数，是这两者的和。图 2-2 对此有说明。

`malloc` 可分配的最大内存是跟系统相关的，看起来这个大小由 `size_t` 限制。不过这个限制可能受可用的物理内存和操作系统的其他限制所影响。

执行 `malloc` 时应该分配所请求数量的内存然后返回内存地址。如果操作系统采用“惰性初始化”策略直到访问内存才真正分配的话会怎样？这时候万一没有足够的内存用来分配就会有问题，答案取决于运行时的操作系统。开发者一般不需要处理这个问题，因为这种初始化策略非常罕见。

5. 静态、全局指针和malloc

初始化静态或全局变量时不能调用函数。下面的代码声明一个静态变量，并试图用 `malloc` 来初始化：

```
static int *pi = malloc(sizeof(int));
```

这样会产生一个编译时错误消息，全局变量也一样。对于静态变量，可以通过在后面用一个单独的语句给变量分配内存来避免这个问题。但是全局变量不能用单独的赋值语句，因为全局变量是在函数和可执行代码外部声明的，赋值语句这类代码必须出现在函数中：

```
static int *pi;
pi = malloc(sizeof(int));
```



在编译器看来，作为初始化操作符的 = 和作为赋值操作符的 = 不一样。

2.2.2　使用calloc函数

`calloc` 会在分配的同时清空内存。该函数的原型如下：

```
void *calloc(size_t numElements, size_t elementSize);
```



清空内存的意思是将其内容置为二进制 0。

calloc 函数会根据 numElements 和 elementSize 两个参数的乘积来分配内存，并返回一个指向内存的第一个字节的指针。如果不能分配内存，则会返回 NULL。此函数最初用来辅助分配数组内存。

如果 numElements 或 elementSize 为 0，那么 calloc 可能返回空指针。如果 calloc 无法分配内存就会返回空指针，而且全局变量 errno 会设置为 ENOMEM（内存不足），这是 POSIX 错误码，有的系统上可能没有。

下例为 pi 分配了 20 字节，全部包含 0：

```
int *pi = calloc(5,sizeof(int));
```

不用 calloc 的话，用 malloc 函数和 memset 函数可以得到同样的结果，如下所示：

```
int *pi = malloc(5 * sizeof(int));
memset(pi, 0, 5* sizeof(int));
```



memset 函数会用某个值填充内存块。第一个参数是指向要填充的缓冲区的指针，第二个参数是填充缓冲区的值，最后一个参数是要填充的字节数。

如果内存需要清零可以使用 calloc，不过执行 calloc 可能比执行 malloc 慢。



cfree 函数已经没用了。早期的 C 用 cfree 来释放 calloc 分配的内存。

2.2.3　使用realloc函数

我们可能需要时不时地增加或减少为指针分配的内存，如果需要一个变长数组这种做法尤其有用，第 4 章会讨论这一点。realloc 函数会重新分配内存，下面是它的原型：

```
void *realloc(void *ptr, size_t size);
```

realloc 函数返回指向内存块的指针。该函数接受两个参数，第一个参数是指向原内存块的指针，第二个是请求的大小。重新分配的块大小和第一个参数引用的块大小不同。返回值是指向重新分配的内存的指针。

请求的大小可以比当前分配的字节数小或者大。如果比当前分配的小，那么多余的内存会还给堆，不能保证多余的内存会被清空。如果比当前分配的大，那么可能的

话，就在紧挨着当前分配内存的区域分配新的内存，否则就会在堆的其他区域分配并把旧的内存复制到新区域。

如果大小是 0 而指针非空，那么就释放内存。如果无法分配空间，那么原来的内存块就保持不变，不过返回的指针是空指针，且 errno 会设置为 ENOMEM。

该函数的行为概括在表 2-2 中。

表2-2：realloc函数的行为

第一个参数	第二个参数	行　　为
空	无	同 malloc
非空	0	原内存块被释放
非空	比原内存块小	利用当前的块分配更小的块
非空	比原内存块大	要么在当前位置要么在其他位置分配更大的块

在下例中，我们使用两个变量为字符串分配内存。一开始分配 16 字节，但只用到了前面的 13 字节（12 个十六进制数字外加 null 结束字符（0））：

```
char *string1;
char *string2;
string1 = (char*) malloc(16);
strcpy(string1, "0123456789AB");
```

接着，用 realloc 函数指定一个范围更小的内存区域。然后打印这两个变量的地址和内容：

```
string2 = realloc(string1, 8);
printf("string1 Value: %p [%s]\n", string1, string1);
printf("string2 Value: %p [%s]\n", string2, string2);
```

输出如下：

```
string1  Value: 0x500 [0123456789AB]
string2  Value: 0x500 [0123456789AB]
```

图 2-6 说明了内存如何分配。

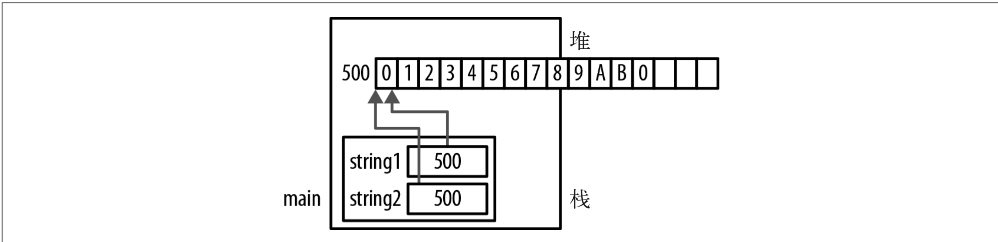


图 2-6：realloc 示例

堆管理器可以重用原始的内存块，且不会修改其内容。不过程序继续使用的内存超过了所请求的 8 字节。也就是说，我们没有修改字符串以便它能装进 8 字节的内存块中。在本例中，我们本应该调整字符串的长度以使它能装进重新分配的 8 字节。实现这一点最简单的办法是将 NUL 赋给地址 507。实际使用的内存超出分配的内存不是个好做法，应该避免，第 7 章会详细讲解这一点。

在接下来的例子中，我们会重新分配额外的内存：

```
string1 = (char*) malloc(16);
strcpy(string1, "0123456789AB");
string2 = realloc(string1, 64);
printf("string1 Value: %p [%s]\n", string1, string1);
printf("string2 Value: %p [%s]\n", string2, string2);
```

执行以上代码得到类似下面的结果：

```
string1  Value: 0x500 [0123456789AB]
string2  Value: 0x600 [0123456789AB]
```

在本例中，`realloc` 必须分配一个新的内存块。图 2-7 说明了内存的分配。

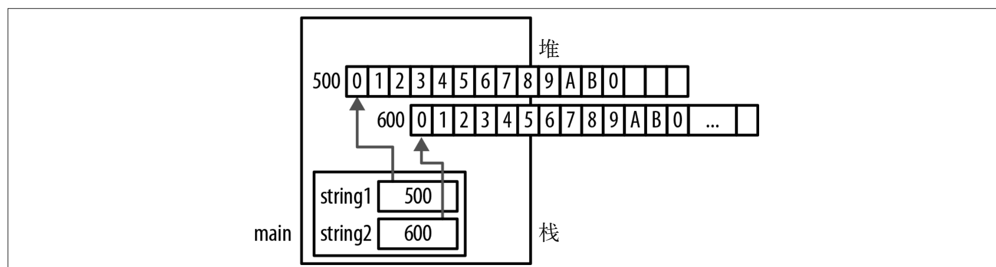


图 2-7：分配额外内存

2.2.4　alloca函数和变长数组

`alloca` 函数（微软为 `malloca`）在函数的栈帧上分配内存。函数返回后会自动释放内存。若底层的运行时系统不基于栈，`alloca` 函数会很难实现，所以这个函数是不标准的，如果应用程序需要可移植就尽量避免使用它。

C99 引入了变长数组（VLA），允许函数内部声明和创建其长度由变量决定的数组。在下例中，我们分配了一个在函数内使用的 `char` 数组：

```
void compute(int size) {
    char* buffer[size];
    ...
}
```

这意味着内存分配在运行时完成，且将内存作为栈帧的一部分来分配。另外，如果数组用到 `sizeof` 操作符，也是在运行时而不是编译时执行。

这么做只会有一点小小的运行时开销。而且一旦函数退出，立即释放内存。因为我们没有用 `malloc` 这类函数来创建数组，所以不应该用 `free` 函数来释放它。`alloca` 函数也不应该返回指向数组所在内存的指针，这个问题在第 5 章解决。



VLA 的长度不能改变，一经分配其长度就固定了。如果你需要一个长度能够实际变化的数组，那么需要使用类似 `realloc` 的函数，2.2.3 节讨论的正是这种方法。

2.3 用free函数释放内存

有了动态内存分配，程序员可以将不再使用的内存返还给系统，这样可以释放内存留作他用。通常用 `free` 函数实现这一点，该函数的原型如下：

```
void free(void *ptr);
```

指针参数应该指向由 `malloc` 类函数分配的内存的地址，这块内存会被返还给堆。尽管指针仍然指向这块区域，但是我们应该将它看成指向垃圾数据。稍后可能重新分配这块区域，并将其装进不同的数据。

在下面这个简单的例子中，`pi` 指向分配的内存，这块内存最终会被释放：

```
int *pi = (int*) malloc(sizeof(int));
...
free(pi);
```

图 2-8 说明了 `free` 函数执行前后瞬间内存的分配情况。地址 500 处的虚线框表示内存已经释放，但仍然有可能包含原值，`pi` 变量仍然指向地址 500。这种情况称为迷途指针，会在 2.4 节详细讨论。

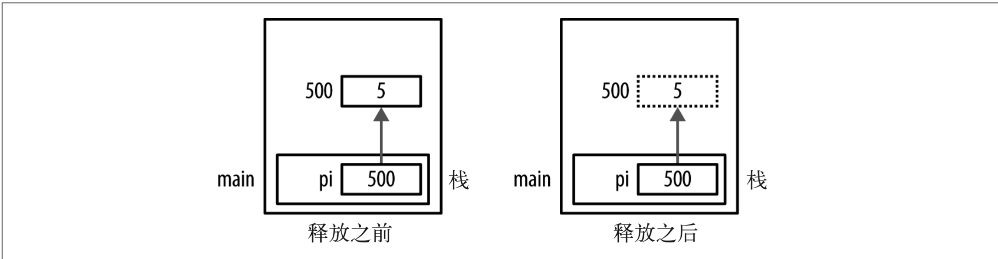


图 2-8：用 free 释放内存

如果传递给 free 函数的参数是空指针，通常它什么都不做。如果传入的指针所指向的内存不是由 malloc 类的函数分配，那么该函数的行为将是未定义的。在下例中，分配给 pi 的是 num 的地址，不过这不是一个合法的堆地址：

```
int num;
int *pi = &num;
free(pi);   // 未定义行为
```



应该在同一层管理内存的分配和释放。比如说，如果是在函数内分配内存，那么就应该在同一个函数内释放它。

2.3.1　将已释放的指针赋值为NULL

已释放的指针仍然可能造成问题。如果我们试图解引一个已释放的指针，其行为将是未定义的。所以有些程序员会显式地给指针赋 NULL 来表示该指针无效，后续再使用这种指针会造成运行时异常。

下面是该方法的示例：

```
int *pi = (int*) malloc(sizeof(int));
...
free(pi);
pi = NULL;
```

内存分配如图 2-9 所示。

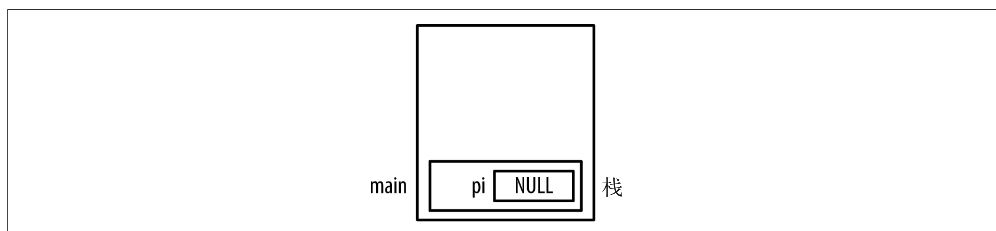


图 2-9：调用 free 后给指针赋值 NULL

这种技术的目的是解决迷途指针类问题。不过，花时间处理造成这类问题的条件要比粗暴地用空指针一刀切好，更何况除了初始化的情况，都不能将 NULL 赋给指针。

2.3.2　重复释放

重复释放是指两次释放同一块内存。下面是一个简单的例子：

```
int *pi = (int*) malloc(sizeof(int));
*pi = 5;
free(pi);
 ...
free(pi);
```

调用第二个 free 函数会导致运行时异常。另一个例子不那么明显，涉及指向同一块内存的两个指针。如下所示，如果我们试图第二次释放同一块内存会发生同样的运行时异常。

```
int *p1 = (int*) malloc(sizeof(int));
int *p2 = p1;
free(p1);
...
free(p2);
```

内存分配如图 2-10 所示。

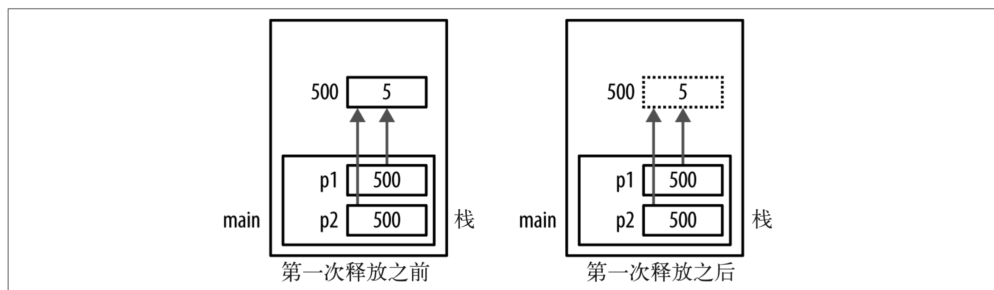


图 2-10：重复释放



两个指针引用同一个地址称为别名，这个概念将在第 8 章讨论。

不幸的是，堆管理器很难判断一个块是否已经被释放，因此它们不会试图去检测是否两次释放了同一块内存。这通常会导致堆损坏和程序终止，即使程序没有终止，它意味着程序逻辑可能存在问题，同一块内存没有理由释放两次。

有人建议 free 函数应该在返回时将 NULL 或其他某个特殊值赋给自身的参数。但指针是传值的，因此 free 函数无法显式地给它赋值 NULL，3.2.7 节会详细讨论这个问题。

2.3.3 堆和系统内存

堆一般利用操作系统的功能来管理内存。堆的大小可能在程序创建后就固定不变了，也可能可以增长。不过堆管理器不一定会在调用 free 函数时将内存返还给操作系

统。释放的内存只是可供应用程序后续使用。所以，如果程序先分配内存然后释放，从操作系统的角度看，释放的内存通常不会反映在应用程序的内存使用上。

2.3.4 程序结束前释放内存

操作系统负责维护应用程序的资源，包括内存。当应用程序终止时，操作系统要负责重新分配这块内存以便别的应用程序使用。已终止的应用程序的内存状态不管是否损坏都无关紧要，事实上，内存损坏可能正是应用程序终止的原因。异常终止的程序可能无法做清理工作，因此没有理由在程序终止之前释放分配的内存。

话虽如此，可能又有一些原因要求我们在程序终止前释放内存。尽责的程序员可能会把释放内存当成质量指标。即使应用程序正在终止，不再使用内存后将其释放总归是好习惯。如果用工具来检测内存泄漏或是类似问题，那么释放内存会让这类工具的输出是干净的。在有些相对简单的操作系统上，操作系统本身可能不会自动回收内存，而是需要程序在终止前回收内存。还有，新版的应用程序可能会在程序末尾增加代码，如果之前的内存没有释放就可能出问题。

因此，确保程序终止前释放所有内存：

- 可能得不偿失；
- 可能很耗时，释放复杂结构也比较麻烦；
- 可能增加应用程序大小；
- 导致更长的运行时间；
- 增加引入更多编程错误的概率。

是否要在程序终止前释放内存取决于具体的应用程序。

2.4 迷途指针

如果内存已经释放，而指针还在引用原始内存，这样的指针就称为迷途指针。迷途指针没有指向有效对象，有时候也称为过早释放。

使用迷途指针会造成一系列问题，包括：

- 如果访问内存，则行为不可预期；
- 如果内存不可访问，则是段错误；
- 潜在的安全隐患。

导致这几类问题的情况可能如下：

- 访问已释放的内存；
- 返回的指针指向的是上次函数调用中的自动变量（在 3.2.5 节中会讨论）。

2.4.1　迷途指针示例

在下面这个简单的例子中我们用 malloc 函数为一个整数分配内存，接下来，用 free 函数释放内存：

```
int *pi = (int*) malloc(sizeof(int));
*pi = 5;
printf("*pi: %d\n", *pi);
free(pi);
```

pi 变量持有整数的地址，但堆管理器可以重复利用这块内存，且其中存放的可能是非整数数据。图 2-11 说明了 free 函数执行前后的程序状态。假设 pi 变量属于 main 函数，位于地址 100，malloc 函数分配的内存位于地址 500。

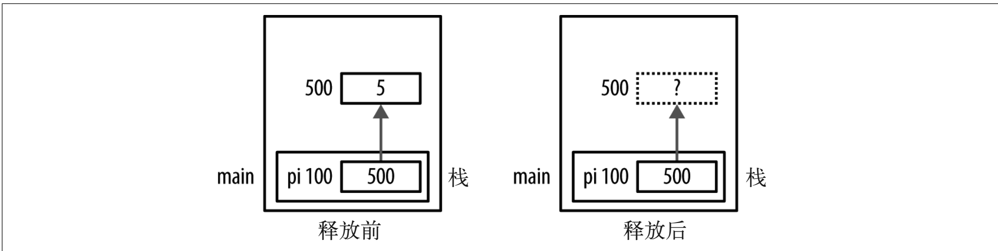


图 2-11：迷途指针

执行 free 函数将释放地址 500 处的内存，此后就不应该再使用这块内存了。但大部分运行时系统不会阻止后续的访问或修改。我们还是可以向这个位置写入数据，如下所示。这么做的结果是不可预期的。

```
free(pi);
*pi = 10;
```

还有一种迷途指针的情况更难觉察：一个以上的指针引用同一内存区域而其中一个指针被释放。如下所示，p1 和 p2 都引用同一块内存区域（称为指针别名），不过 p1 被释放了：

```
int *p1 = (int*) malloc(sizeof(int));
*p1 = 5;
...
int *p2;
p2 = p1;
...
free(p1);
```

```
...
*p2 = 10;    // 迷途指针
```

图 2-12 说明了内存分配情况，虚线框表示释放的内存。

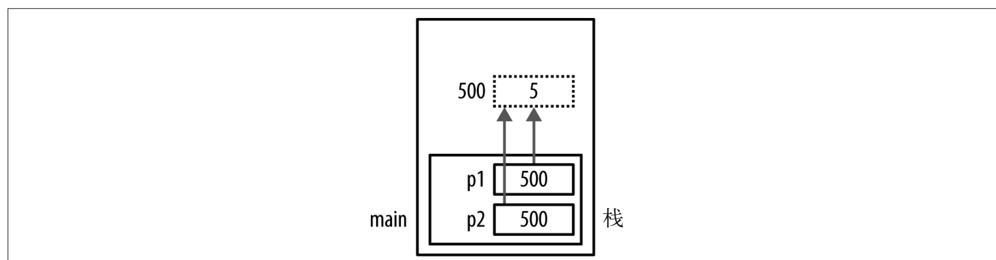


图 2-12　迷途指针和指针别名

使用块语句时也可能出现一些小问题，如下所示。这里 pi 被赋值为 tmp 的地址，变量 pi 可能是全局变量，也可能是局部变量。不过当包含 tmp 的块出栈之后，地址就不再有效：

```
int *pi;
...
{
   int tmp = 5;
   pi = &tmp;
}
// 这里 pi 变成了迷途指针
foo();
```

大部分编译器都把块语句当做一个栈帧。tmp 变量分配在栈帧上，之后在块语句退出时会出栈。pi 指针现在指向一块最终可能被其他活跃记录（比如 foo 函数）覆盖的内存区域。图 2-13 说明的就是这种情形。

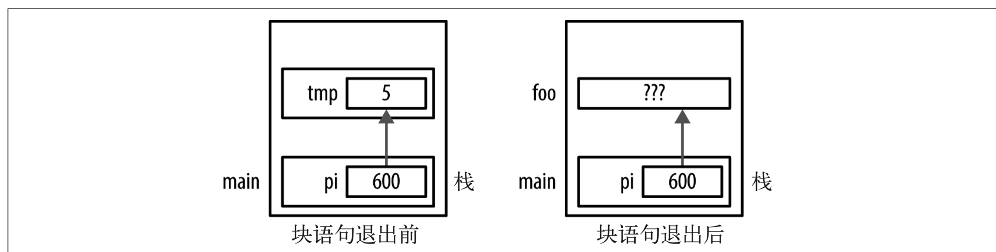


图 2-13：块语句的问题

2.4.2　处理迷途指针

有时候调试指针诱发的问题会很难解决，以下方法可用来对付迷途指针。

- 释放指针后置为 `NULL`，后续使用这个指针会终止应用程序。不过，如果存在多个指针的话还是会有问题。因为赋值只会影响一个指针，2.3.2 节中有相关说明。
- 写一个特殊的函数代替 `free` 函数（参见 3.2.7 节）。
- 有些系统（运行时或调试系统）会在释放后覆写数据（比如 0xDEADBEEF，取决于被释放的对象，Visual Studio 会用 0xCC、0xCD 或者 0xDD）。在不抛出异常的情况下，如果程序员在预期之外的地方看到这些值，可以认为程序可能在访问已释放的内存。
- 用第三方工具检测迷途指针和其他问题。

在调试迷途指针时打印指针的值可能会有所帮助，但需要注意打印的方式。1.1.5 节已经讨论过如何打印指针的值。确保用一致的方式打印，从而避免比较指针的值时产生歧义。`assert` 宏也可能有用，7.1.3 节中会讲到。

2.4.3 调试器对检测内存泄漏的支持

微软提供了解决动态分配内存的覆写和内存泄漏的技术。这种方法在调试版程序里用了特殊的内存管理技术：

- 检查堆的完整性；
- 检查内存泄漏；
- 模拟堆内存不够的情况。

微软是通过使用一种特殊的数据结构管理内存分配来做到这一点的。这种结构维护调试信息，比如 `malloc` 调用点的文件名和行号，还会在实际的内存分配之前和之后分配缓冲区来检测对实际内存的覆写。关于这种技术的更多信息可以参考 Microsoft Developer Network（http://msdn.microsoft.com/en-us/library/x98tx3cf.aspx）。

Mudflap 库（http://gcc.fyxm.net/summit/2003/mudflap.pdf）为 GCC 编译器提供了类似的功能，它的运行时库支持对内存泄漏的检测和其他功能，这种检测是通过监控指针解引操作来实现的。

2.5 动态内存分配技术

目前为止，我们已经讨论了如何使用堆管理器分配和释放内存。不过，不同的编译器在技术实现上有所不同。大部分堆管理器把堆或数据段作为内存资源。这种方法的缺点是会造成碎片，而且可能和程序栈碰撞。尽管如此，它还是实现堆最常用的方法。

堆管理器需要处理很多问题，比如堆是否基于进程和（或）线程分配，如何保护堆

不受安全攻击。

堆管理器有不少，包括OpenBSD的malloc、Hoard的malloc和Google开发的TCMalloc。GNU C库的分配器基于通用分配器dlmalloc（http://dmalloc.com），它提供调试机制，能追踪内存泄漏。dlmalloc的日志特性可以追踪内存的使用和内存事务，还有一些其他功能。

6.3节会讲到手动管理结构体内存的技术。

2.5.1 C的垃圾回收

`malloc`和`free`函数提供了手动分配和释放内存的方法。不过对于很多问题，需要考虑使用C的手动内存管理，比如性能、达到好的引用局部性、线程问题，以及优雅地清理内存。

有些非标准的技术可以用来解决部分问题，本节将探讨其中一部分技术。这些技术的关键特性在于自动释放内存。内存不再使用之后会被收集起来以备后续使用，释放的内存称为垃圾，因此，垃圾回收就是指这个过程。

鉴于以下原因，垃圾回收是有价值的：

- 不需要程序员费尽心思决定何时释放内存；
- 让程序员专注应用程序本身的问题。

Boehm-Weiser Collector（http://www.hpl.hp.com/personal/Hans_Boehm/gc/）可以作为手动内存管理的替换方法，不过它不属于语言的一部分。

2.5.2 资源获取即初始化

资源获取即初始化（Resource Acquisition Is Initialization，RAII）是Bjarne Stroustrup发明的技术，可以用来解决C++中资源的分配和释放。即使有异常发生，这种技术也能保证资源的初始化和后续的释放。分配的资源最终总是会得到释放。

有好几种方法可以在C中使用RAII。GNU编译器提供了非标准的扩展来支持这个特性，通过演示如何在一个函数中分配内存然后释放可以说明这种扩展。一旦变量超出作用域会自动触发释放过程。

GNU的扩展要用到`RAII_VARIABLE`宏，它声明一个变量，然后给变量关联如下属性：

- 一个类型；

- 创建变量时执行的函数；
- 变量超出作用域时执行的函数。

这个宏如下所示：

```
#define RAII_VARIABLE(vartype,varname,initval,dtor) \
    void _dtor_ ## varname (vartype * v) { dtor(*v); } \
    vartype varname __attribute__((cleanup(_dtor_ ## varname))) = (initval)
```

在下例中，我们将 `name` 变量声明为字符指针。创建它时会执行 `malloc` 函数，为其分配 32 字节。当函数结束时，`name` 超出作用域就会执行 `free` 函数：

```
void raiiExample() {
    RAII_VARIABLE(char*, name, (char*)malloc(32), free);
    strcpy(name,"RAII Example");
    printf("%s\n",name);
}
```

函数执行后会打印 `"RAII_Example"` 字符串。

不用 GNU 扩展也可以达到类似的效果（http://en.wikipedia.org/wiki/Resource_Acquisition_Is_ Initialization#Ad-hoc_mechanisms）。

2.5.3 使用异常处理函数

另一种处理内存释放的方法是利用异常处理（http://www.adomas.org/excc/）。尽管异常处理不属于标准 C，但如果可以使用它且不考虑移植问题，它会很有用。下面说明利用 Microsoft Visual Studio 版的 C 语言的方法。

这里的 `try` 块包含任何可能在运行时抛出异常的语句。不管有没有异常抛出，都会执行 `finally` 块，因此也一定会执行 `free` 函数。

```
void exceptionExample() {
   int *pi = NULL;
   __try {
      pi = (int*)malloc(sizeof(int));
      *pi = 5;
      printf("%d\n",*pi);
   }
   __finally {
      free(pi);
   }
}
```

也可以用别的方法在 C 中实现异常处理。

2.6 小结

动态内存分配是 C 语言的重要特性。本章主要关注用 `malloc` 和 `free` 函数实现手动分配内存。我们解决了涉及这两个函数的一系列常见问题，包括内存分配失败和迷途指针。

此外，还有一些非标准技术可用来实现 C 的动态内存管理。我们也接触了几种垃圾回收技术，包括 RAII 和异常处理。

第3章

指针和函数

指针对函数功能的贡献极大。它们能够将数据传递给函数，并且允许函数对数据进行修改。我们可以将复杂数据用结构体指针的形式传递给函数和从函数返回。如果指针持有函数的地址，就能动态控制程序的执行流。在本章中，我们会探索指针与函数结合使用的能力，学习如何用指针解决很多真实存在的问题。

要理解函数及其和指针的结合使用，需要理解程序栈。大部分现代的块结构语言，比如 C，都用到了程序栈来支持函数执行。调用函数时，会创建函数的栈帧并将其推到程序栈上。函数返回时，其栈帧从程序栈上弹出。

在使用函数时，有两种情况指针很有用。首先是将指针传递给函数，这时函数可以修改指针所引用的数据，也可以更高效地传递大块信息。

另一种情况是声明函数指针。本质上，函数表示法就是指针表示法。函数名字经过求值会变成函数的地址，然后函数参数会被传递给函数。我们将会看到，函数指针为控制程序的执行流提供了新的选择。

下面这一节将为理解和使用函数以及指针打好基础。鉴于函数和指针的普及程度，有这个基础会对你有很大帮助。

3.1 程序的栈和堆

程序的栈和堆是 C 重要的运行时元素。在本节中，我们将仔细研究程序栈和堆的结构以及用法，还会看一下栈帧的结构，它用于保存局部变量。



局部变量也称为自动变量，它们总是分配在栈帧上。

3.1.1 程序栈

程序栈是支持函数执行的内存区域，通常和堆共享。也就是说，它们共享同一块内存区域。程序栈通常占据这块区域的下部，而堆用的则是上部。

程序栈存放*栈帧*（stack frame），栈帧有时候也称为*活跃记录*（activation record）或*活跃帧*（activation frame）。栈帧存放函数参数和局部变量。堆管理动态内存，已经在 2.1 节中讨论过了。

图 3-1 从原理上说明了栈和堆的结构。这个说明基于以下代码片段。

```
void function2() {
    Object *var1 = ...;
    int var2;
    printf("Program Stack Example\n");
}

void function1() {
    Object *var3 = ...;
    function2();
}

int main() {
    int var4;
    function1();
}
```

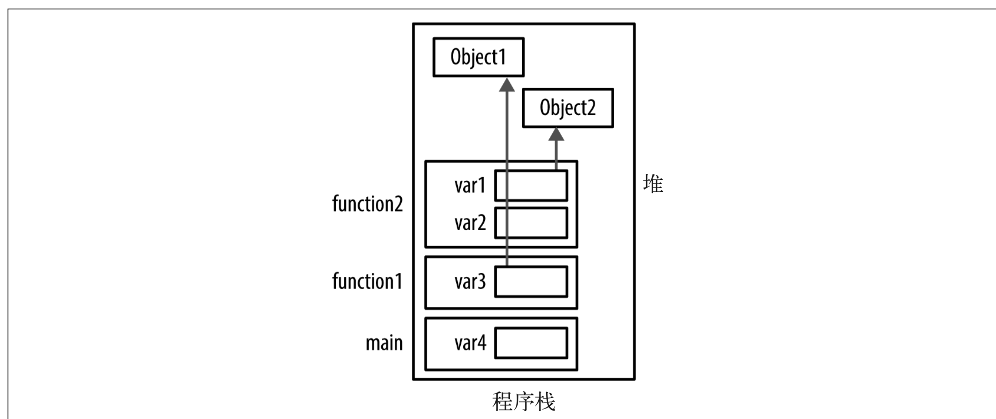


图 3-1：栈和堆

调用函数时，函数的栈帧被推到栈上，栈向上“长出”一个栈帧。当函数终止时，其栈帧从程序栈上弹出。栈帧所使用的内存不会被清理，但最终可能会被推到程序栈上的另一个栈帧覆盖。

动态分配的内存来自堆，堆向下“生长”。随着内存的分配和释放，堆中会布满碎片。尽管堆是向下生长的，但这只是个大体方向，实际上内存可能在堆上的任意位置分配。

3.1.2　栈帧的组织

栈帧由以下几种元素组成。

- *返回地址*
  函数完成后要返回的程序内部地址。
- *局部数据存储*
  为局部变量分配的内存。
- *参数存储*
  为函数参数分配的内存。
- *栈指针和基指针*
  运行时系统用来管理栈的指针。

普通 C 程序员不会关心支持栈帧的栈指针和基指针。不过，理解它们的概念和用法能让你更深入地理解程序栈。

栈指针通常指向栈顶部。基指针（帧指针）通常存在并指向栈帧内部的地址，比如返回地址，用来协助访问栈帧内部的元素。这两个指针都不是 C 指针，它们是运行时系统管理程序栈的地址。如果运行时系统用 C 实现，这些指针倒真是 C 指针。

我们以下面这个函数为例来了解栈帧的创建。该函数传递了一个整数数组和一个表示数组长度的整数。三个 `printf` 语句用来打印参数和局部变量的地址：

```
float average(int *arr, int size) {
   int sum;
   printf("arr: %p\n",&arr);
   printf("size: %p\n",&size);
   printf("sum: %p\n",&sum);

   for(int i=0; i<size; i++) {
       sum += arr[i];
   }
   return (sum * 1.0f) / size;
}
```

执行上述代码会得到类似下面的输出：

```
arr: 0x500
size: 0x504
sum: 0x480
```

参数地址和局部变量地址之间的空档，保存的是运行时系统管理栈所需要的其他栈帧元素。

系统在创建栈帧时，将参数以跟声明时相反的顺序推到帧上，最后推入局部变量，如图 3-2 所示。在这个例子中，`size` 在 `arr` 之前被推入。通常，接下来会推入函数调用的返回地址，然后是局部变量。推入它们的顺序跟其在代码中列出的顺序相反。

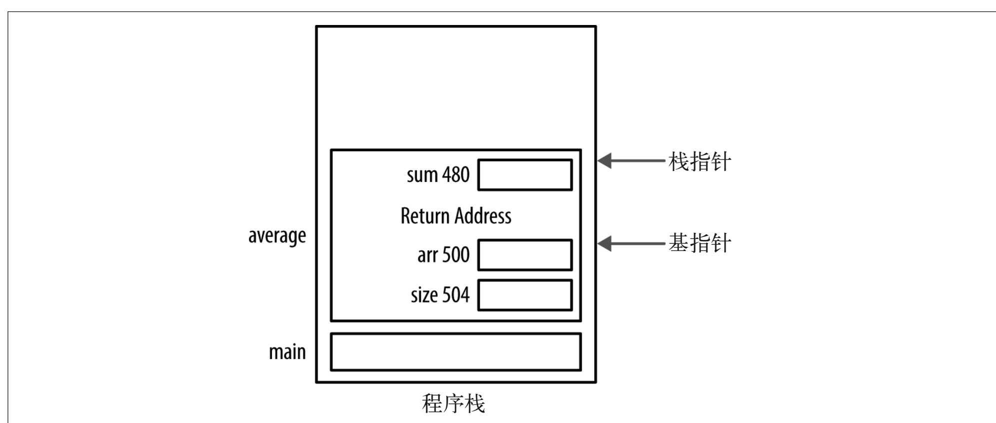


图 3-2：栈帧示例

从原理上说，本例中的栈“向上”生长。不过栈帧的参数和局部变量以及新栈帧被添加到了低内存地址。栈的实际生长方向跟实现相关。

`for` 语句中用到的变量 `i` 没有包含在栈帧中。C 把块语句当做“微型”函数，会在合适的时机将其推入栈和从栈中弹出。在本例中，块语句在执行时被推到程序栈中 `average` 栈帧上面，执行完后又弹出。

精确的地址可能会变化，不过顺序一般不变。这一点很重要，因为它可以解释参数和变量内存分配的相对顺序。在调试指针问题时这一点会很有用。如果你不知道栈帧如何分配，这些地址在你看来也毫无意义。

将栈帧推到程序栈上时，系统可能会耗尽内存，这种情况称为栈溢出，通常会导致程序非正常终止。要牢记每个线程通常都会有自己的程序栈。一个或多个线程访问内存中的同一个对象可能会导致冲突，我们将在 8.3.1 节中讨论这个问题。

3.2 通过指针传递和返回数据

本节讨论将指针传递给函数和从函数返回指针。传递指针可以让多个函数访问指针所引用的对象，而不用把对象声明为全局可访问。这意味着只有需要访问这个对象的函数才有访问权限，而且也不需要复制对象。

要在某个函数中修改数据，需要用指针传递数据。通过传递一个指向常量的指针，可以使用指针传递数据并禁止其被修改，正如 3.2.3 节中所展示的那样。当数据是需要被修改的指针时，我们就传递指针的指针，这个话题在 3.2.7 节中讨论。

传递参数（包括指针）时，传递的是它们的值。也就是说，传递给函数的是参数值的一个副本。当涉及大型数据结构时，传递参数的指针会更高效。比如说一个表示雇员的大型结构体，如果我们把整个结构体传递给函数，那么需要复制结构体的所有字节，这样会导致程序运行变慢，栈帧也会占用过多内存。传递对象的指针意味着不需要复制对象，但可以通过指针访问对象。

3.2.1 用指针传递数据

用指针来传递数据的一个主要原因是函数可以修改数据。下面的代码段实现了一个交换函数，可以交换其参数所引用的值。这是很多排序算法中的常用操作。我们在这里用整数指针，通过解引它们来实现交换。

```
void swapWithPointers(int* pnum1, int* pnum2) {
    int tmp;
    tmp = *pnum1;
    *pnum1 = *pnum2;
    *pnum2 = tmp;
}
```

下面的代码段说明了这个函数的用法：

```
int main() {
    int n1 = 5;
    int n2 = 10;
    swapWithPointers(&n1, &n2);
    return 0;
}
```

指针 `pnum1` 和 `pnum2` 在交换操作中被解引，结果是修改了 `n1` 和 `n2` 的值。图 3-3 说明了内存如何组织，左图表示 `swap` 函数开始时程序栈的状态，而右图则是函数返回前的状态。

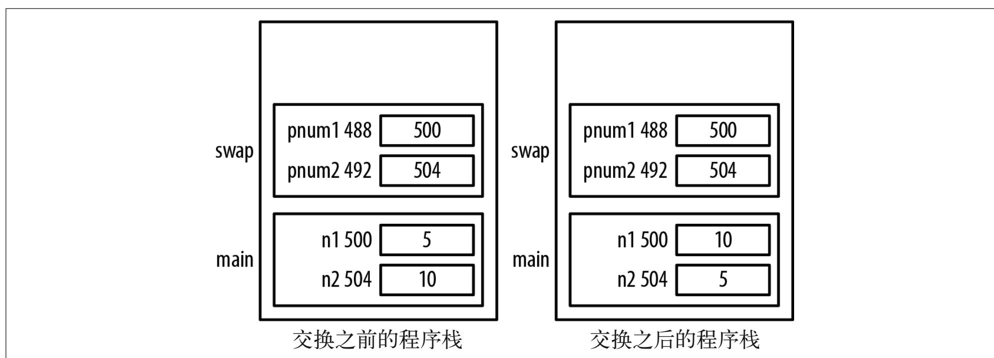


图 3-3：用指针实现交换

3.2.2 用值传递数据

如果不通过指针传递参数，那么交换就不会发生。下面的函数通过值来传递两个整数：

```
void swap(int num1, int num2) {
    int tmp;
    tmp = num1;
    num1 = num2;
    num2 = tmp;
}
```

下面的代码将两个整数传递给函数：

```
int main() {
    int n1 = 5;
    int n2 = 10;
    swap(n1, n2);
    return 0;
}
```

然而，这样并没有实现交换，因为整数是通过值而不是指针来传递的。num1 和 num2 中保存的只是实参的副本。修改 num1，实参 n1 不会变化。修改形参不会影响实参。图 3-4 说明了形参的内存分配。

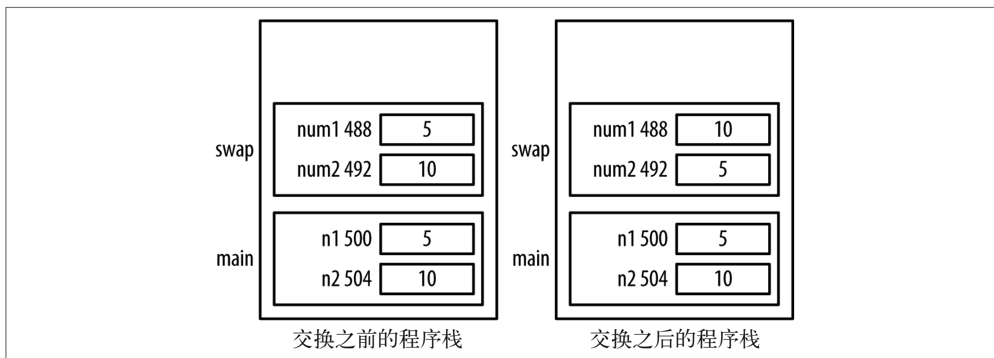


图 3-4：通过值传递数据

3.2.3 传递指向常量的指针

传递指向常量的指针是 C 中常用的技术，效率很高，因为我们只传了数据的地址，能避免某些情况下复制大量内存。不过，如果只是传递指针，数据就能被修改。如果不希望数据被修改，就要传递指向常量的指针。

在本例中，我们传递一个指向整数常量的指针和一个指向整数的指针。在函数内，我们不能修改通过指向常量的指针传进来的值：

```
void passingAddressOfConstants(const int* num1, int* num2) {
    *num2 = *num1;
}

int main() {
    const int limit = 100;
    int result = 5;
    passingAddressOfConstants(&limit, &result);
    return 0;
}
```

这样不会产生语法错误，函数会把 100 赋给 `result` 变量。在下面这个版本的函数中，我们试图修改两个被引用的整数：

```
void passingAddressOfConstants(const int* num1, int* num2) {
    *num1 = 100;
    *num2 = 200;
}
```

如果我们把 `limit` 常量传递给函数的两个参数就会导致问题：

```
const int limit = 100;
passingAddressOfConstants(&limit, &limit);
```

这样会产生一个语法错误，抱怨第二个形参和实参的类型不匹配。此外，它还会抱怨我们试图修改第一个参数所引用的常量。

该函数期待一个整数指针，但是传进来的却是指向整数常量的指针。我们不能把一个整数常量的地址传递给一个指向整数的指针，因为这样会允许修改常量。1.4.2 节有详细讨论。

像下面这样试图传递一个整数字面量的地址也会产生语法错误：

```
passingAddressOfConstants(&23, &23);
```

这种情况错误信息会指出取地址操作符的操作数需要的是一个左值。左值的概念在 1.1.6 节中讨论过了。

3.2.4 返回指针

返回指针很容易，只要返回的类型是某种数据类型的指针即可。从函数返回对象时经常用到以下两种技术。

- 使用 `malloc` 在函数内部分配内存并返回其地址。调用者负责释放返回的内存。
- 传递一个对象给函数并让函数修改它。这样分配和释放对象的内存都是调用者的责任。

首先，我们介绍用 `malloc` 这类函数来分配返回的内存，随后的示例中我们返回一个局部对象的指针，不推荐后一种方法。上面列出的第二种技术在 3.2.6 节中说明。

在下面的例子中，我们定义一个函数，为其传递一个整数数组的长度和一个值来初始化每个元素。函数为整数数组分配内存，用传入的值进行初始化，然后返回数组地址：

```
int* allocateArray(int size, int value) {
    int* arr = (int*)malloc(size * sizeof(int));
    for(int i=0; i<size; i++) {
        arr[i] = value;
    }
    return arr;
}
```

下面说明如何使用这个函数：

```
int* vector = allocateArray(5,45);
for(int i=0; i<5; i++) {
    printf("%d\n", vector[i]);
}
```

图 3-5 说明了这个函数的内存分配。左图显示 `return` 语句执行前的程序状态，右图显示函数返回后的程序状态。`vector` 变量包含了函数内分配的内存的地址。当函数终止时 `arr` 变量也会消失，但是指针所引用的内存还在，这部分内存最终需要释放。

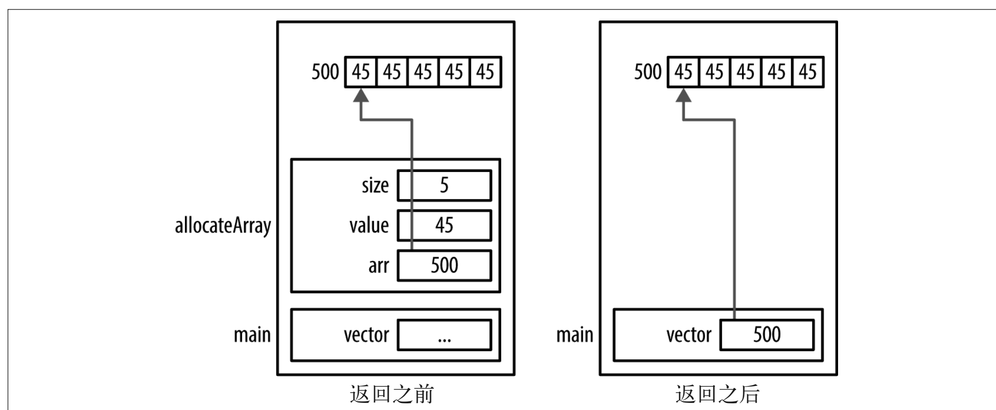


图 3-5：返回指针

尽管上例可以正确工作，但从函数返回指针时可能存在几个潜在的问题，包括：

- 返回未初始化的指针；
- 返回指向无效地址的指针；
- 返回局部变量的指针；
- 返回指针但是没有释放内存。

最后一个问题的典型代表就是 `allocateArray` 函数。从函数返回动态分配的内存意味着函数的调用者有责任释放内存。看一下这个例子：

```
int* vector = allocateArray(5,45);
...
free(vector);
```

最终我们必须在用完后释放内存，否则就会产生内存泄漏。

3.2.5 局部数据指针

如果你不理解程序栈如何工作，就很容易犯返回指向局部数据的指针的错误。在下面的例子中，我们重写了 3.2.4 节中用到的 `allocateArray` 函数。这次我们不为数组动态分配内存，而是用了一个局部数组：

```
int* allocateArray(int size, int value) {
    int arr[size];
    for(int i=0; i<size; i++) {
        arr[i] = value;
    }
    return arr;
}
```

不幸的是，一旦函数返回，返回的数组地址也就无效了，因为函数的栈帧从栈中弹出了。尽管每个数组元素仍然可能包含 45，但如果调用另一个函数，就可能覆写这些值。下面的代码段对此做了演示，重复调用 `printf` 函数导致数组损坏：

```
int* vector = allocateArray(5,45);
for(int i=0; i<5; i++) {
    printf("%d\n", vector[i]);
}
```

图 3-6 说明了发生这种情况时内存的分配状态。虚线框表示其他的栈帧（比如 `printf` 函数用到的），可能会被推到程序栈上，从而损坏数组持有的内存。栈帧的实际内容取决于实现。

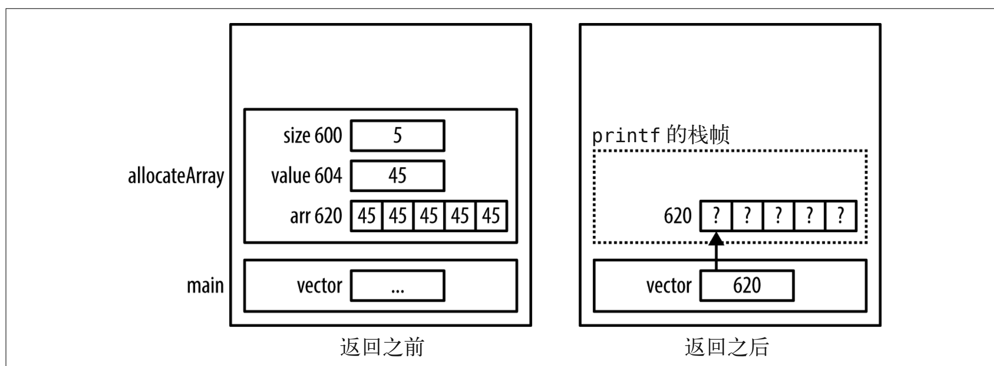


图 3-6：返回局部数据的指针

还有一种方法是把 arr 变量声明为 static。这样会把变量的作用域限制在函数内部，但是分配在栈帧外面，避免其他函数覆写变量值。

```
int* allocateArray(int size, int value) {
    static int arr[5];
    ...
}
```

不过这种方法并不一定总是有用。每次调用 allocateArray 函数都会重复利用这个数组。这样相当于每次都把上一次调用的结果覆盖掉。此外，静态数组必须声明为固定长度，这样会限制函数处理变长数组的能力。

如果函数只是返回一个可能的值，而且共享这些值也不会有什么坏处，那么它可以维护一个这些值的列表，然后返回合适的值。如果我们需要返回状态类型的消息，比如不大可能被修改的错误码，这么做就很有用。5.4 节中有使用全局和静态值的例子。

3.2.6　传递空指针

下面这个版本的 allocateArray 函数传递了一个数组指针、数组的长度和用来初始化数组元素的值。返回指针只是为了方便。这个版本的函数不会分配内存，但后面的版本会分配：

```
int* allocateArray(int *arr, int size, int value) {
    if(arr != NULL) {
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
    return arr;
}
```

将指针传递给函数时，使用之前先判断它是否为空是个好习惯。

该函数可以像这样调用：

```
int* vector = (int*)malloc(5 * sizeof(int));
allocateArray(vector,5,45);
```

如果指针是 NULL，那么什么都不会发生，程序继续执行，不会非正常终止。

3.2.7 传递指针的指针

将指针传递给函数时，传递的是值。如果我们想修改原指针而不是指针的副本，就需要传递指针的指针。在下例中，我们传递了一个整数数组的指针，为该数组分配内存并将其初始化。函数会用第一个参数返回分配的内存。在函数中，我们先分配内存，然后初始化。所分配的内存地址应该被赋给一个整数指针。为了在调用函数中修改这个指针，我们需要传入指针的地址。所以，参数被声明为 int 指针的指针。在调用函数中，我们需要传递指针的地址：

```
void allocateArray(int **arr, int size, int value) {
    *arr = (int*)malloc(size * sizeof(int));
    if(*arr != NULL) {
        for(int i=0; i<size; i++) {
            *(*arr+i) = value;
        }
    }
}
```

这个函数可以用下面的代码测试：

```
int *vector = NULL;
allocateArray(&vector,5,45);
```

allocateArray 的第一个参数以整数指针的指针的形式传递。当我们调用这个函数时，需要传递这种类型的值。这是通过传递 vector 地址做到的。malloc 返回的地址被赋给 arr。解引整数指针的指针得到的是整数指针。因为这是 vector 的地址，所以我们修改了 vector。

内存分配说明如图 3-7 所示。左图显示 malloc 返回且初始化数组后的栈状态。类似地，右图显示函数返回后的栈状态。

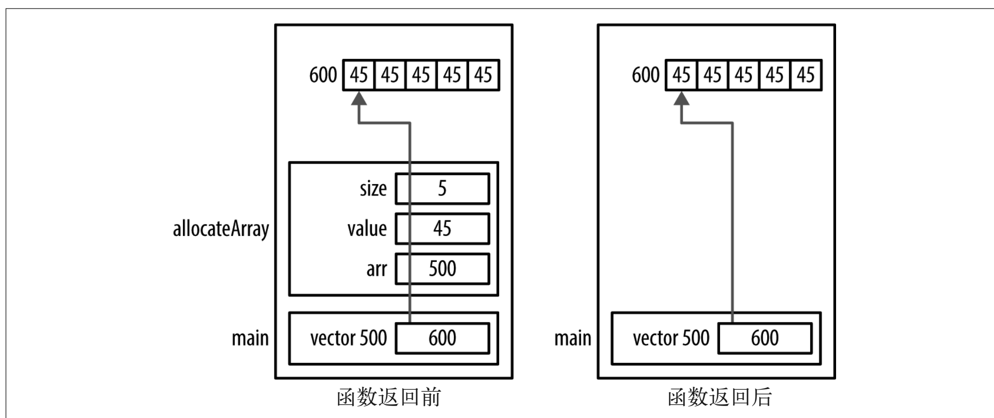


图 3-7：传递指针的指针



要方便地发现内存泄漏这样的问题，只需画一张内存分配图。

下面这个版本的函数说明了为什么只传递一个指针不会起作用：

```
void allocateArray(int *arr, int size, int value) {
    arr = (int*)malloc(size * sizeof(int));
    if(arr != NULL) {
        for(int i=0; i<size; i++) {
            arr[i] = value;
        }
    }
}
```

下面的代码段说明了如何使用这个函数：

```
int *vector = NULL;
allocateArray(vector,5,45);
printf("%p\n",vector);
```

运行后会看到程序打印出 0x0，因为将 vector 传递给函数时，它的值被复制到了参数 arr 中，修改 arr 对 vector 没有影响。当函数返回后，没有将存储在 arr 中的值复制到 vector 中。图 3-8 说明了内存分配情况：左图显示 arr 被赋新值之前的内存状态；中图显示 allocateArray 函数中的 malloc 函数执行且初始化数组后的内存状态，arr 变量被修改为指向堆中的某个新位置；右图显示函数返回后程序栈的状态。此外，这里有内存泄漏，因为我们无法再访问地址 600 处的内存块了。

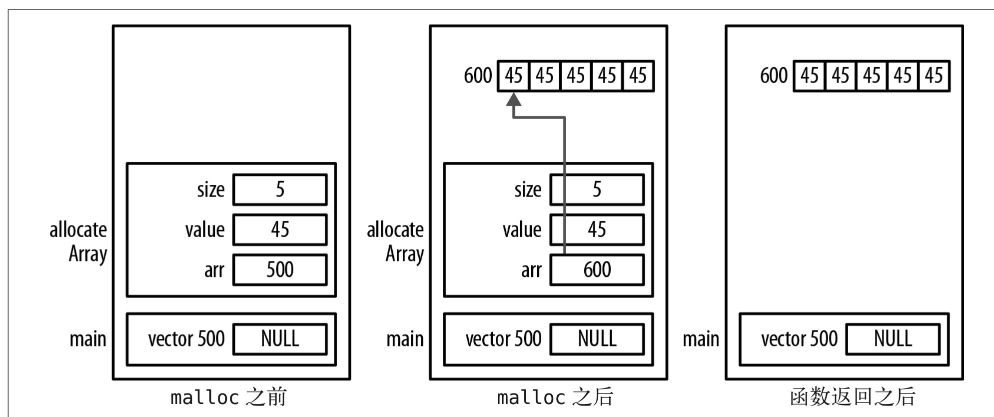


图 3-8：传递指针

实现自己的free函数

由于 free 函数存在一些问题，因而某些程序员创建了自己的 free 函数。free 函数不会检查传入的指针是否是 NULL，也不会在返回前把指针置为 NULL。释放指针之后将其置为 NULL 是个好习惯。

有了 3.2 节中的基础知识，我们给出下面这个 free 函数的实现，可以给指针赋 NULL。此处需要我们给它传递一个指针的指针：

```
void saferFree(void **pp) {
    if (pp != NULL && *pp != NULL) {
        free(*pp);
        *pp = NULL;
    }
}
```

saferFree 函数调用实际释放内存的 free 函数，前者的参数声明为 void 指针的指针。使用指针的指针允许我们修改传入的指针，而使用 void 类型则可以传入所有类型的指针。不过，如果调用这个函数时没有显式地把指针类型转换为 void 会产生警告，执行显式转换就不会有警告。

下面这个 safeFree 宏调用 saferFree 函数，执行类型转换，并使用了取地址操作符，这样就省去了函数使用者做类型转换和传递指针的地址：

```
#define safeFree(p) saferFree((void**)&(p))
```

下面的代码片段说明了这个宏的用法：

```
int main() {
    int *pi;
    pi = (int*) malloc(sizeof(int));
    *pi = 5;
    printf("Before: %p\n",pi);
    safeFree(pi);
    printf("After: %p\n",pi);
    safeFree(pi);
    return (EXIT_SUCCESS);
}
```

假设 `malloc` 返回的内存位于地址 1000，那么这段代码的输出是 1000 和 0。第二次调用 `safeFree` 宏给它传递 `NULL` 值不会导致程序终止，因为 `saferFree` 函数检测到这种情况并忽略了这个操作。

3.3　函数指针

函数指针是持有函数地址的指针。指针能够指向函数对于 C 来说是很重要也很有用的特性，这为我们以编译时未确定的顺序执行函数提供了另一种选择，而不需要使用条件语句。

人们使用函数指针的一个顾虑是这种做法可能会导致程序运行变慢，处理器可能无法配合流水线做分支预测。分支预测是处理器用来推测哪块代码会被执行的技术。流水线是常用的提升处理器性能的硬件技术，通过重叠指令的执行来实现。在这种机制下，处理器会处理它认为能执行的分支，如果预测正确，那么就不需要丢弃当前流水线中的指令。

函数指针对性能的影响要视具体情况而定。在表查找等场景中使用函数指针可以缓解性能问题。在本节中，我们会学习如何声明函数指针，如何使用函数指针来支持其他的执行路径，还会探索能够充分发挥函数指针潜能的技术。

3.3.1　声明函数指针

第一次看到声明函数指针的语法时你可能会感到迷惑。不过跟 C 的很多方面一样，一旦熟悉这种表示法，理解起来就顺理成章了。下面我们声明一个函数指针，该函数接受空参数，返回空值。

```
void (*foo)();
```

这个声明很像函数原型声明。如果去掉第一对括号，看起来就像函数 `foo` 的原型，它接受 `void`，返回 `void` 指针。不过，括号让这个声明变成了一个名为 `foo` 的函数指针。星号表示这是个指针。图 3-9 说明了函数指针声明的各个部分。

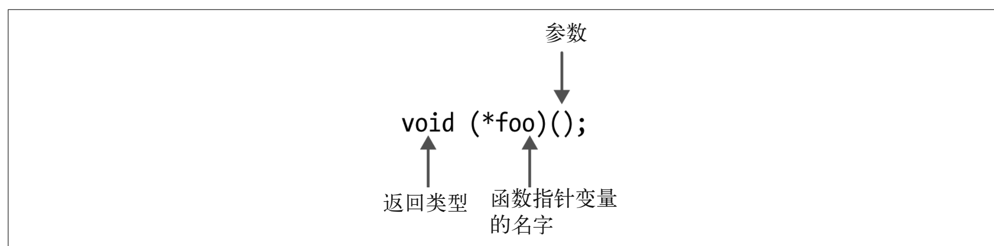


图 3-9：函数指针声明



使用函数指针时一定要小心，因为 C 不会检查参数传递是否正确。

下面是声明函数指针的其他一些例子：

```
int (*f1)(double);          // 传入 double，返回 int
void (*f2)(char*);          // 传入 char 指针，没有返回值
double* (*f3)(int, int);    // 传递两个整数，返回 double 指针
```



我们对函数指针在命名约定上的建议是用 `fptr` 做前缀。

不要把返回指针的函数和函数指针搞混。下面的 `f4` 是一个函数，它返回一个整数指针，而 `f5` 是一个返回整数的函数指针，变量 `f6` 是一个返回整数指针的函数指针。

```
int  *f4();
int  (*f5)();
int* (*f6)();
```

可以调整这些表达式中的空白符，如下所示：

```
int*  f4();
int (*f5)();
```

很明显，`f4` 是个返回整数指针的函数，而 `f5` 的括号则明确地把表示“指针”的星号和函数名绑定在一起，所以它是个函数指针。

3.3.2　使用函数指针

下面是使用函数指针的一个简单示例，其中函数接受一个整数参数并返回一个整数。我们也定义了 `square` 函数，对一个整数求平方并返回值。为了简化例子，假定整数不会溢出。

```
int (*fptr1)(int);

int square(int num) {
    return num*num;
}
```

要用函数指针来调用 square 函数，需要把 square 函数的地址赋给函数指针，如下所示。就像数组名字一样，我们用的是函数本身的名字，它会返回函数的地址。我们还声明了一个整数并将其传递给函数：

```
int n = 5;
fptr1 = square;
printf("%d squared is %d\n",n, fptr1(n));
```

执行代码后会显示“5 squared is 25.”。我们也可以像下面那样用取地址操作符对函数名进行操作，但是没必要这么做。在这种上下文环境中编译器会忽略取地址操作符。

```
fptr1 = &square;
```

图 3-10 说明了本例的内存分配。我们把 square 函数放在程序栈下方。这只是举例子，实际上函数会被分配在跟程序栈所用段不同的段上。函数的实际地址通常对我们没用。

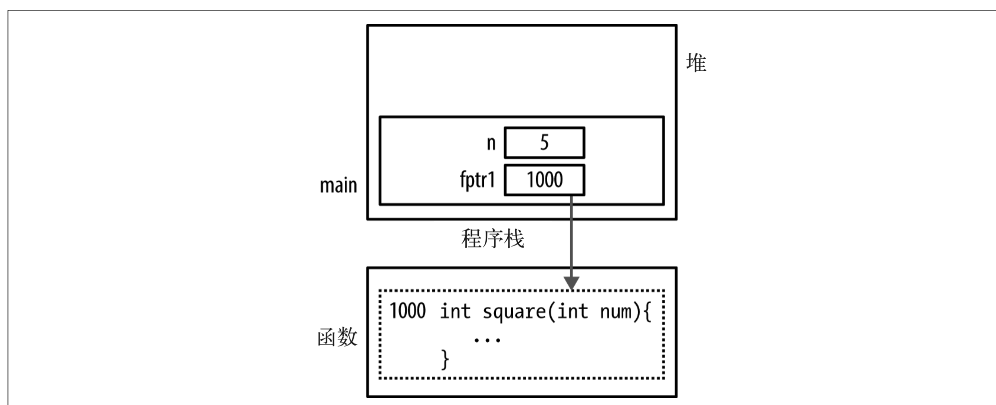


图 3-10：函数的位置

为函数指针声明一个类型定义会比较方便，下面说明对于之前用到的函数指针应该怎么做。类型定义看起来有点奇怪，通常，类型定义的名字是声明的最后一个元素。

```
typedef int (*funcptr)(int);

...
```

```
funcptr fptr2;
fptr2 = square;
printf("%d squared is %d\n",n, fptr2(n));
```

5.5 节提供了一个有趣的例子，讲的是用函数指针来控制字符串的排序方式。

3.3.3 传递函数指针

传递函数指针很简单，只要把函数指针声明作为函数参数即可。我们会用下面这个例子中的 add、sub 和 compute 函数来说明如何传递函数指针：

```
int add(int num1, int num2) {
   return num1 + num2;
}

int subtract(int num1, int num2) {
   return num1 - num2;
}

typedef int (*fptrOperation)(int,int);

int compute(fptrOperation operation, int num1, int num2) {
   return operation(num1, num2);
}
```

下面的代码片段说明如何使用这些函数：

```
printf("%d\n",compute(add,5,6));
printf("%d\n",compute(subtract,5,6));
```

输出是 11 和 −1。add 和 sub 函数的地址被传递给 compute 函数，后者使用这些地址来调用对应的操作。本例也说明了使用函数指针可以让代码变得更灵活。

3.3.4 返回函数指针

返回函数指针需要把函数的返回类型声明为函数指针，为了说明如何实现这一点，我们会沿用 3.3.3 节中的 add 和 sub 函数，以及类型定义。

我们用下面的 select 函数基于输入的字符来返回一个指向对应操作的函数指针。取决于传入的操作码，它要么返回 add 函数，要么返回 sub 函数。

```
fptrOperation select(char opcode) {
    switch(opcode) {
        case '+': return add;
        case '-': return subtract;
    }
}
```

evaluate 函数把这些函数联系在一起，该函数接受两个整数和一个字符，字符代表要做的操作，它会把 opcode 传递给 select 函数，后者返回要执行的函数指针。在返回语句中，evaluate 函数执行刚才返回的函数并返回结果。

```
int evaluate(char opcode, int num1, int num2) {
    fptrOperation operation = select(opcode);
    return operation(num1, num2);
}
```

evaluate 函数及 printf 语句的用法如下所示：

```
printf("%d\n",evaluate('+', 5, 6));
printf("%d\n",evaluate('-', 5, 6));
```

输出是 11 和 −1。

3.3.5 使用函数指针数组

函数指针数组可以基于某些条件选择要执行的函数，声明这种数组很简单，只要把函数指针声明为数组的类型即可，如下所示。这个数组的所有元素都被初始化为 NULL。如果数组的初始化值是一个语句块，系统会将块中的值赋给连续的数组元素。本例中只有一个值，我们会用这个值来初始化数组的所有元素。

```
typedef int (*operation)(int, int);
operation operations[128] = {NULL};
```

也可以不用 typedef 来声明这个数组，如下：

```
int (*operations[128])(int, int) = {NULL};
```

这个数组的目的是可以用一个字符索引选择对应的函数来执行。比如，如果存在 * 字符就表示乘法函数，我们可以用字符作为索引是因为字符字面量其实是整数，128 个元素对应前 128 个 ASCII 字符。我们会把这个定义用在 3.3.4 节中实现的 add、sub 函数上。

数组初始化为 NULL 后，我们把 add 和 sub 函数赋给加号和减号对应的元素：

```
void initializeOperationsArray() {
    operations['+'] = add;
    operations['-'] = subtract;
}
```

将前面的 evaluate 函数改写为 evaluateArray。接下来我们用操作字符作为索引来使用 operations，而不是调用 select 函数来获取函数指针。

```
int evaluateArray(char opcode, int num1, int num2) {
    fptrOperation operation;
    operation = operations[opcode];
    return operation(num1, num2);
}
```

用下面的代码测试这些函数：

```
initializeOperationsArray();
printf("%d\n",evaluateArray('+', 5, 6));
printf("%d\n",evaluateArray('-', 5, 6));
```

执行结果是 11 和 −1。更健壮的 `evaluateArray` 函数版本需要在执行函数之前检查空指针。

3.3.6 比较函数指针

我们可以用相等和不等操作符来比较函数指针。下例中用到了 `fptrOperation` 类型定义和 3.3.3 节中的 `add` 函数。`add` 函数被赋给 `fptr1` 函数指针，然后和 `add` 函数的地址做比较：

```
fptrOperation fptr1 = add;

if(fptr1 == add) {
    printf("fptr1 points to add function\n");
} else {
    printf("fptr1 does not point to add function\n");
}
```

执行这段代码后，通过输出可以看到指针确实指向了 `add` 函数。

可以说明比较函数指针用处的一个更现实的例子是，用函数指针数组表示一系列任务步骤的情况。比如说，我们可能会有一系列函数维护一个库存部件数组。可能用一组操作来对部件排序，计算总数，然后打印出数组和总数；用另一组操作打印数组，找到最贵和最便宜的部件，然后显示差额。每种操作都可以用指向各自函数的指针的数组来表示。日志操作可能同时出现在上述两组操作中。借助比较两个函数指针，我们可以通过删除某个操作（比如日志）来动态修改操作，只要从列表中找到并删除对应的函数指针即可。

3.3.7 转换函数指针

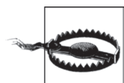
我们可以将指向某个函数的指针转换为其他类型的指针，不过要谨慎使用，因为运行时系统不会验证函数指针所用的参数是否正确。也可以把一种函数指针转换为另一种再转换回来，得到的结果和原指针相同，但函数指针的长度不一定相等。下面

的代码说明了这个操作：

```
typedef int (*fptrToSingleInt)(int);
typedef int (*fptrToTwoInts)(int,int);
int add(int, int);

fptrToTwoInts fptrFirst = add;
fptrToSingleInt fptrSecond = (fptrToSingleInt)fptrFirst;
fptrFirst = (fptrToTwoInts)fptrSecond;
printf("%d\n",fptrFirst(5,6));
```

这段代码执行后输出 11。



无法保证函数指针和数据指针相互转换后正常工作。

void* 指针不一定能用在函数指针上，也就是说我们不应该像下面这样把函数指针赋给 void* 指针：

```
void* pv = add;
```

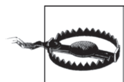
不过在交换函数指针时，通常会见到如下声明所示的“基本”函数指针类型。这里把 fptrBase 声明为指向不接受参数也不返回结果的函数的函数指针。

```
typedef void (*fptrBase)();
```

下面的代码片段说明了基本指针的用法，跟上一个例子是一样的效果：

```
fptrBase basePointer;
fptrFirst = add;
basePointer = (fptrToSingleInt)fptrFirst;
fptrFirst = (fptrToTwoInts)basePointer;
printf("%d\n",fptrFirst(5,6));
```

基本指针用做占位符，用来交换函数指针的值。



一定要确保给函数指针传递正确的参数，否则会造成不确定的行为。

3.4　小结

理解程序栈和堆有助于更深入彻底地理解程序的工作方式以及指针的行为。在本章中，我们研究了栈、堆和栈帧，这些概念对解释将指针传递给函数和从函数返回指针的机制有帮助。

比如说，返回指向局部变量的指针是错误的，原因是为局部变量分配的内存会在后续的函数调用中被覆盖。传递指向常量数据的指针很高效，还可以防止函数修改传入的数据。传递指针的指针可以让参数指针指向不同的内存地址，栈和堆可以帮助深入解释这些功能。

我们也介绍和讲解了函数指针，这类指针允许应用程序根据需要执行不同的函数，对于控制应用程序内的执行序列很有用。

第4章

指针和数组

数组是C内建的基本数据结构，彻底理解数组及其用法是开发高效应用程序的基础。曲解数组和指针的用法会造成难以查找的错误，应用程序的性能也难以达到最优。数组和指针表示法紧密关联，在合适的上下文中可以互换。

一种常见的错误观点是数组和指针是完全可以互换的。尽管数组名字有时候可以当做指针来用，但数组的名字不是指针。数组表示法也可以和指针一起使用，但两者明显不同，也不一定能互换。理解这种差别可以帮助你避免错误地使用这些表示法。比如说，尽管数组使用自身的名字可以返回数组地址，但是名字本身不能作为赋值操作的目标。

数组在应用程序中随处可见，可能是一维，也可能是多维。在本章中，我们会讲解数组跟指针相关的基础知识，以便你深入理解数组以及使用指针操作数组的各种方法。本书其他章节还会展示在更高级的环境中使用数组和指针。

本章首先概述数组，然后研究数组表示法和指针表示法的相同点和不同点。可以用`malloc`类函数创建数组，这些函数提供比传统的数组声明更灵活的机制。我们会看到如何用`realloc`函数来改变已经为一个数组分配的内存大小。

为数组动态分配内存可以为代码带来很大的改变，特别是处理二维或多维数组的情况，因为我们得确保为数组分配的内存是连续的。

我们也会探索传递和返回数组时可能发生的问题。大部分情况下，必须传入数组长度以便函数正确处理数组。数组的内部表示不带有长度信息，如果我们不传递长度，

函数就没有标准的方法得到数组的终点。即便并不常用，我们也会研究如何在 C 中创建不规则数组。不规则数组是二维数组，每一行都可能包含不同的列数。

要说明这些概念，需要使用向量和矩阵，前者代表一维数组，后者代表二维数组。向量和矩阵用途广泛，包括电磁场分析、天气预报和数学上的应用。

4.1 数组概述

数组是能用索引访问的同质元素连续集合。这里所说的连续是指数组的元素在内存中是相邻的，中间不存在空隙，而同质是指元素都是同一类型的。数组声明用的是方括号集合，可以拥有多个维度。

二维数组很常见，我们一般用行和列来表述数组元素的位置。三维或更多维的数组不是很常见，不过有些应用程序会用到。不要混淆二维数组和指针的数组，它们很类似，但是行为有点差别，我们会在 4.6 节中讲到。

C99 标准引入了变长数组，在此之前，支持变长数组的技术是用 `realloc` 函数实现的。我们会在 4.4 节中说明 `realloc` 函数。



数组的长度是固定的，当我们声明数组时，需要决定该数组有多大。如果指定过多元素就会浪费空间，而指定过少元素就会限制能够处理的元素数量。`realloc` 函数和变长数组提供了应对长度需要变化的数组的技术。只要略施小计，我们就能调整数组长度，只占用合适的内存。

4.1.1 一维数组

一维数组是线性结构，用一个索引访问成员。下面的代码声明了一个 5 个元素的整数数组：

```
int vector[5];
```

数组索引从 0 开始，到声明的长度减 1 结束。`vector` 数组的索引从 0 开始，到 4 结束。不过，C 并没有强制规定边界，用无效的索引访问数组会造成不可预期的行为。图 4-1 说明了数组的内存如何分配，每个元素 4 字节长，且没有初始化。就像 1.2.1 节中所解释的，取决于不同的内存模型，数组的长度可能会不同。

数组的内部表示不包含其元素数量的信息，数组名字只是引用了一块内存。对数组做 `sizeof` 操作会得到为该数组分配的字节数，要知道元素的数量，只需将数组长度除以元素长度，如下所示，打印结果是 5：

```
printf("%d\n", sizeof(vector)/sizeof(int));
```

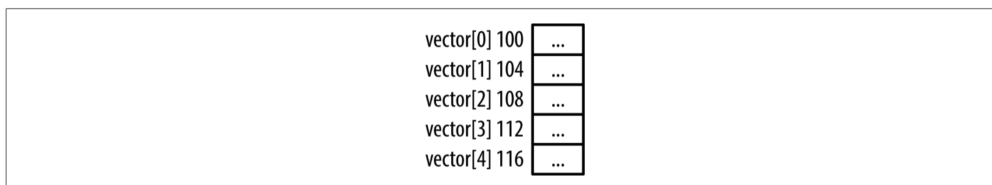


图 4-1：数组的内存分配

可以用一个块语句初始化一维数组，下面的代码把数组中的元素初始化为从 1 开始的整数：

```
int vector[5] = {1, 2, 3, 4, 5};
```

4.1.2　二维数组

二维数组使用行和列来标识数组元素，这类数组需要映射为内存中的一维地址空间。在 C 中这是通过行－列顺序实现的。先将数组的第一行放进内存，接着是第二行、第三行，直到最后一行。

下面声明了一个 2 行 3 列的二维数组，用块语句对数组进行了初始化。图 4-2 说明了这个数组的内存分配，左图说明内存如何映射，右图显示数组在概念上的样子。

```
int matrix[2][3] = {{1,2,3},{4,5,6}};
```

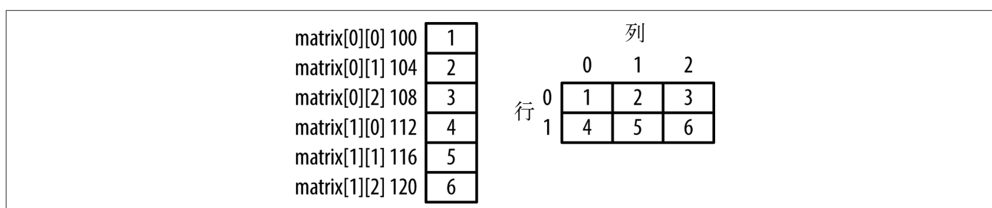


图 4-2：二维数组

我们可以将二维数组当做数组的数组，也就是说，如果只用一个下标访问数组，得到的是对应行的指针。下面的代码说明了这个概念，它会打印每一行的地址和长度：

```
for (int i = 0; i < 2; i++) {
    printf("&matrix[%d]: %p  sizeof(matrix[%d]): %d\n",
            i, &matrix[i], i, sizeof(matrix[i]));
}
```

下面的输出假设数组位于地址 100，因为每行有 3 个元素，每个元素 4 字节长，所

以数组长度是 12：

```
&matrix[0]: 100 sizeof(matrix[0]): 12
&matrix[1]: 112 sizeof(matrix[1]): 12
```

在 4.7 节中我们会深入研究这种行为。

4.1.3　多维数组

多维数组具有两个及两个以上维度。对于多维数组，需要多组括号来定义数组的类型和长度。下面的例子中，我们定义了一个具有 3 行、2 列、4 阶的三维数组。阶通常用来标识第三维元素。

```
int arr3d[3][2][4] = {
    {{1, 2, 3, 4}, {5, 6, 7, 8}},
    {{9, 10, 11, 12}, {13, 14, 15, 16}},
    {{17, 18, 19, 20}, {21, 22, 23, 24}}
};
```

元素按照行－列－阶的顺序连续分配，如图 4-3 所示。

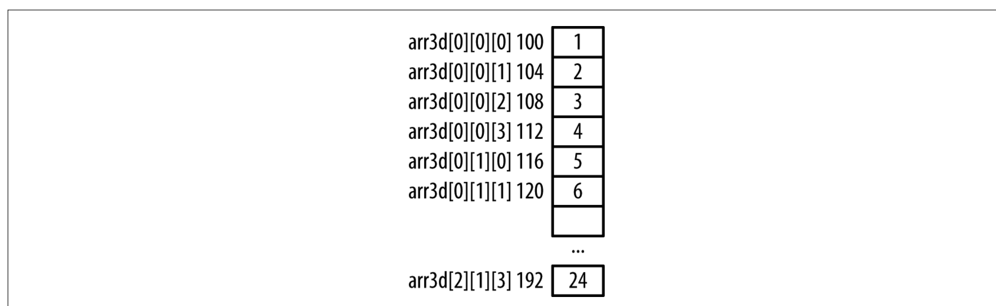


图 4-3：三维数组

我们会在后面的例子中用到这些声明。

4.2　指针表示法和数组

指针在处理数组时很有用，我们可以用指针指向已有的数组，也可以从堆上分配内存然后把这块内存当做一个数组使用。数组表示法和指针表示法在某种意义上可以互换。不过，它们并不完全相同，后面的“数组和指针的差别”中会详细说明。

单独使用数组名字时会返回数组地址。我们可以把地址赋给指针，如下所示：

```
int vector[5] = {1, 2, 3, 4, 5};
int *pv = vector;
```

pv 变量是指向数组第一个元素而不是指向数组本身的指针。给 pv 赋值是把数组的第一个元素的地址赋给 pv。

我们可以只用数组名字，也可以对数组的第一个元素用取地址操作符，如下所示。这些写法是等价的，都会返回 vector 的地址。用取地址操作符更繁琐一些，不过也更明确。

```
printf("%p\n",vector);
printf("%p\n",&vector[0]);
```

有时候也会使用 &vector 这个表达式获取数组的地址，不同于其他表示法，这么做返回的是整个数组的指针，其他两种方法得到是整数指针。这种类型的用法会在 4.8 节解释。

我们可以把数组下标用在指针上，实际上，pv[i] 这种表示法等价于：

```
*(pv + i)
```

pv 指针包含一个内存块的地址，方括号表示法会取出 pv 中包含的地址，用指针算术运算把索引 i 加上，然后解引新地址返回其内容。

就像我们在 1.3.1 节中讨论的那样，给指针加上一个整数会把它持有的地址增加这个整数和数据类型长度的乘积，这一点对于给数组名字加上整数也适用。下面两个语句是等价的：

```
*(pv + i)
*(vector + i)
```

假设 vector 位于地址 100，pv 位于地址 96，表 4-1 和图 4-4 说明了如何利用数组下标和指针算术运算分别从数组名字和指针得到不同的值。

表4-1：数组/指针表示法

值	等价表达式			
92	&vector[-2]	vector - 2	&pv[-2]	pv - 2
100	vector	vector + 0	&pv[0]	pv
100	&vector[0]	vector + 0	&pv[0]	pv
104	&vector[1]	vector + 1	&pv[1]	pv + 1
140	&vector[10]	vector + 10	&pv[10]	pv + 10

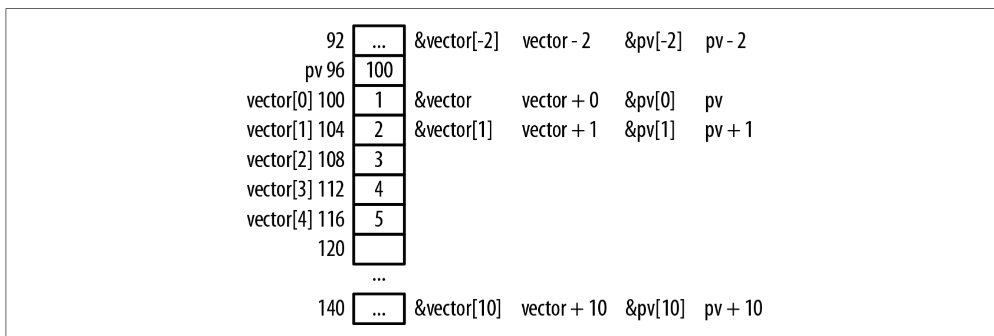


图 4-4：数组 / 指针表示法

给数组地址加 1 实际加了 4，也就是整数的长度，因为这是一个整数数组。对于第一个和最后一个操作，我们越过了数组边界，这不是好习惯，不过也提醒我们在用索引和指针访问数组元素时要谨慎。

数组表示法可以理解为“偏移并解引”操作。vector[2] 表达式表示从 vector 开始，向右偏移两个位置，然后解引那个位置获取其值，其中 vector 是指向数组起始位置的指针。如果用取地址操作符和数组表示法，就像 &vector[-2]，其实就是去掉了解引操作，可以解释为向左偏移两个位置然后返回地址。

下面的代码说明了标量相加操作的实现中指针的使用。这个操作接受一个值然后给 vector 的每个元素乘上这个值：

```
pv = vector;
int value = 3;
for(int i=0; i<5; i++) {
    *pv++ *= value;
}
```

数组和指针的差别

数组和数组指针在使用上有一些区别，本节使用的 vector 数组和 pv 指针定义如下：

```
int vector[5] = {1, 2, 3, 4, 5};
int *pv = vector;
```

vector[i] 生成的代码和 *(vector+i) 生成的不一样，vector[i] 表示法生成的机器码从位置 vector 开始，移动 i 个位置，取出内容。而 *(vector+i) 表示法生成的机器码则是从 vector 开始，在地址上增加 i，然后取出这个地址中的内容。尽管结果是一样的，生成的机器码却不一样，对于大部分人来说，这种差别几乎无足轻重。

sizeof 操作符对数组和同一个数组的指针操作也是不同的。对 vector 调用 sizeof 操作符会返回 20，就是这个数组分配的字节数。对 pv 调用 sizeof 操作符会返回 4，就是指针的长度。

pv 是一个左值，左值表示赋值操作符左边的符号。左值必须能修改。像 vector 这样的数组名字不是左值，它不能被修改。我们不能改变数组所持有的地址，但可以给指针赋一个新值从而引用不同的内存段。

考虑如下代码：

```
pv = pv + 1;
vector = vector + 1; // 语法错误
```

我们无法修改 vector，只能修改它的内容。不过，vector+1 表达式本身没问题，如下所示：

```
pv = vector + 1;
```

4.3 用malloc创建一维数组

如果从堆上分配内存并把地址赋给一个指针，那就肯定可以对指针使用数组下标并把这块内存当成一个数组。在下面的代码中，我们复制之前用过的数组 vector 中的内容：

```
int *pv = (int*) malloc(5 * sizeof(int));
for(int i=0; i<5; i++) {
    pv[i] = i+1;
}
```

也可以像下面这样使用指针表示法，不过数组表示法通常更简单：

```
for(int i=0; i<5; i++) {
    *(pv+i) = i+1;
}
```

图 4-5 说明了本例的内存分配。

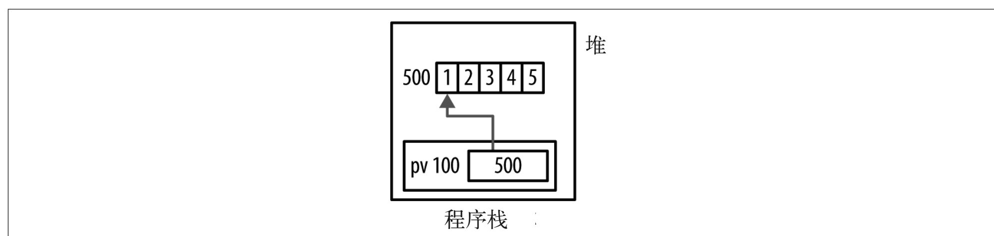
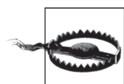


图 4-5：从堆上分配数组

这种技术分配一块内存并把它当成数组，其长度在运行时确定。不过，我们得记得用完之后释放内存。



在上个例子中我们用的是 `*(pv+i)` 而不是 `*pv+i`，因为解引操作符的优先级比加操作符高，先解引第二个表达式的指针，得到指针所引用的值，然后再给这个整数加上 `i`。这不是我们要的效果，而且，如果我们把这个表达式作为左值，编译器会抱怨。所以，为了让代码正确工作，我们需要强制先做加法，然后才是解引操作。

4.4 用realloc调整数组长度

用 `malloc` 创建的已有数组的长度可以通过 `realloc` 函数来调整。`realloc` 函数的基本知识已经在第 2 章详细探讨过了。C99 标准支持变长数组，有些情况下这种解决方案可能比使用 `realloc` 函数更好。如果没有使用 C99，那就只能用 `realloc`。此外，变长数组只能在函数内部声明，如果数组需要的生命周期比函数长，那也只能用 `realloc`。

为了说明 `realloc` 函数，我们会实现一个从标准输入读取字符并放入缓冲区的函数，缓冲区会包含除最后的回车字符之外的所有字符。我们无法得知用户会输入多少字符，因此也就无法知道缓冲区应该有多长。我们会用 `realloc` 函数通过一个定长增量来分配额外空间。实现该函数的代码如下所示：

```
char* getLine(void) {
    const size_t sizeIncrement = 10;
    char* buffer = malloc(sizeIncrement);
    char* currentPosition = buffer;
    size_t maximumLength = sizeIncrement;
    size_t length = 0;
    int character;

    if(currentPosition == NULL) { return NULL; }

    while(1) {
        character = fgetc(stdin);
        if(character == '\n') { break; }

        if(++length >= maximumLength) {
            char *newBuffer = realloc(buffer, maximumLength += sizeIncrement);

            if(newBuffer == NULL) {
                free(buffer);
                return NULL;
            }
```

```
            currentPosition = newBuffer + (currentPosition - buffer);
            buffer = newBuffer;
        }
        *currentPosition++ = character;
    }
    *currentPosition = '\0';
    return buffer;
}
```

首先我们声明了一系列变量，总结在表 4-2 中。

表4-2：getLine函数的变量

sizeIncrement	缓冲区的初始大小以及需要增大时的增量
buffer	指向读入字符的指针
currentPosition	指向缓冲区中下一个空白位置的指针
maximumLength	可以安全地存入缓冲区的最大字符数
length	读入的字符数
character	上次读入的字符数

缓冲区创建时的大小是 sizeIncrement，如果 malloc 函数无法分配内存，第一个 if 语句会强制 getLine 函数返回 NULL。接着是一次处理一个字符的无限循环，循环退出后，字符串末尾会添加上 NUL，然后返回缓冲区的地址。

在 while 循环内部，程序每次读入一个字符，如果是回车符，循环退出。接着，if 语句判断我们有没有超出缓冲区大小，如果没有超出，字符就被添加到缓冲区中。

如果超出了缓冲区大小，realloc 函数会分配一块新内存，这块内存比旧内存大 sizeIncrement 字节。如果无法分配内存，我们会释放现有的已分配内存，强制函数返回 NULL；否则 currentPosition 会调整为指向新分配的缓冲区。realloc 函数不一定会让已有的内存保持在原来的位置，所以必须用它返回的指针来确定调整过大小的内存块的位置。

newBuffer 变量持有已分配内存的地址，我们需要用别的变量而不是 buffer，这样万一 realloc 无法分配内存，我们也可以检测到这种情况并进行处理。

如果 realloc 分配成功，我们不需要释放 buffer，因为 realloc 会把原来的缓冲区复制到新的缓冲区中，再把旧的释放。如果试图释放 buffer，十有八九程序会终止，因为我们试图重复释放同一块内存。

图 4-6 说明了 getLine 函数面对 Once upon a time there was a giant pumpkin 这个输入字符串时的内存分配情况。我们简化了程序栈，省略了除 buffer 和 currentPosition 之外的局部变量。根据包含字符串的方框来看，buffer 增长了四次。

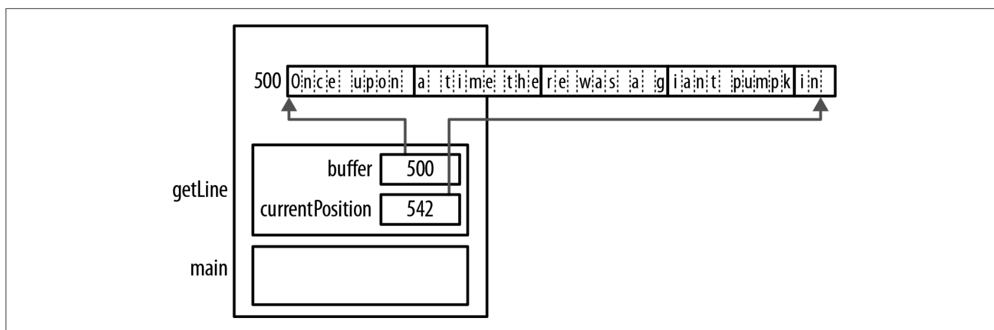


图 4-6：getLine 函数的内存分配

realloc 函数也可以用来减少指针指向的内存。为了说明这种用法，如下所示的 trim 函数会把字符串中开头的空白符删掉：

```
char* trim(char* phrase) {
    char* old = phrase;
    char* new = phrase;

    while(*old == ' ') {
        old++;
    }

    while(*old) {
        *(new++) = *(old++);
    }
    *new = 0;
    return (char*) realloc(phrase,strlen(phrase)+1);
}

int main() {
    char* buffer = (char*)malloc(strlen("  cat")+1);
    strcpy(buffer," cat");
    printf("%s\n",trim(buffer));
}
```

第一个 while 循环跳过开头的空白符，第二个 while 循环把字符串中剩下的字符复制到字符串的开头，它的判断条件一直是真，直到遇到 NUL 字符，就会变成假，接着字符串末尾会添加 0。然后我们会根据字符串的长度用 realloc 函数调整内存大小。

图 4-7 说明了该函数接受 " cat" 字符串作为参数时的执行情况。字符串在 trim 函数执行前后的状态如图所示，红色框内的内存是旧内存，不应该访问。

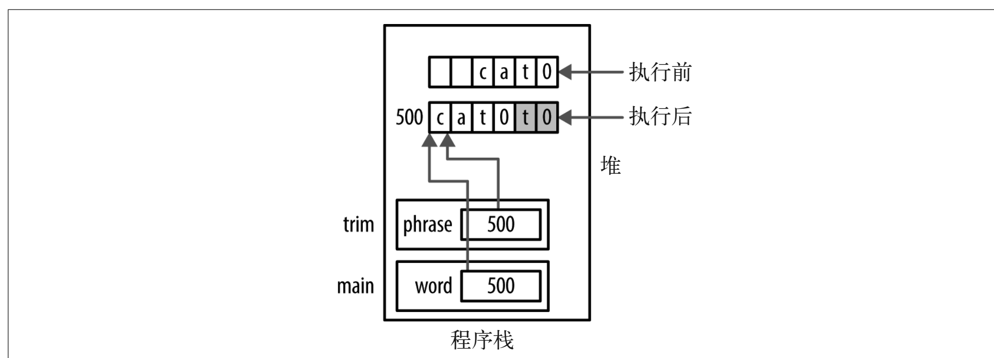


图 4-7：realloc 示例

4.5 传递一维数组

将一维数组作为参数传递给函数实际是通过值来传递数组的地址，这样信息传递就更高效，因为我们不需要传递整个数组，从而也就不需要在栈上分配内存。通常，这也意味着要传递数组长度，否则在函数看来，我们只有数组的地址而不知道其长度。

除非数组内部有信息告诉我们数组的边界，否则在传递数组时也需要传递长度信息。如果数组内存储的是字符串，我们可以依赖 NUL 字符来判断何时停止处理数组，第 5 章会深入探讨这部分内容。一般来说，如果不知道数组长度，就无法处理其元素，最终导致的结果可能是处理的元素太少，也可能是把数组边界以外的内存当成数组的一部分，而这样经常会造成程序非正常终止。

我们可以使用下面两种表示法中的一种在函数声明中声明数组：数组表示法和指针表示法。

4.5.1 用数组表示法

下面的例子将一个整数数组及其长度传递给函数，并打印其内容：

```
void displayArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d\n", arr[i]);
    }
}

    int vector[5] = {1, 2, 3, 4, 5};
    displayArray(vector, 5);
```

这段代码的输出是数字 1 到 5，我们给函数传递 5 来表明数组长度。也可以传递任意正数，不管这个长度是否正确，函数都会试图打印相应数量的元素。尝试越过数

组边界寻址可能会导致程序终止。本例的内存分配如图 4-8 所示。

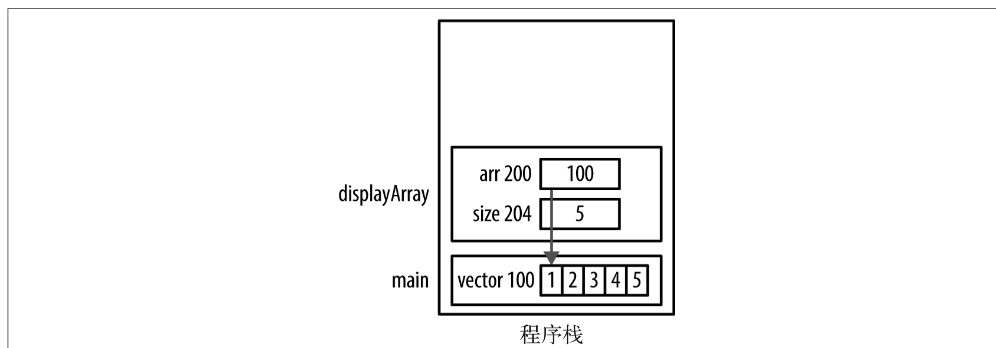
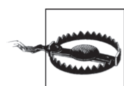


图 4-8：使用数组表示法



为确定数组的元素数量对数组使用 `sizeof` 操作符是一种常见的错误，如下所示。4.1.1 节中已经解释过了，这样获取长度是不对的。在这种情况下，我们给函数传递的是 20。

```
displayArray(arr, sizeof(arr));
```

还有一种情况比较常见：传递的元素数量比数组中实际的元素数量少，这样可以处理数组的一部分。比如说，假设我们读入一系列年龄并放进数组，但没有占满数组，此时如果调用 `sort` 函数来排序，我们希望只对有效的年龄进行排序，而不是数组的所有元素。

4.5.2 用指针表示法

声明函数的数组参数不一定要用方括号表示法，也可以用指针表示法，如下所示：

```
void displayArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d\n", arr[i]);
    }
}
```

在函数内部我们仍然使用数组表示法，如果有需要，也可以用指针表示法：

```
void displayArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d\n", *(arr+i));
    }
}
```

如果在声明函数时用了数组表示法，在函数体内还是可以用指针表示法：

```
void displayArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d\n", *(arr+i));
    }
}
```

4.6 使用指针的一维数组

在本节中，我们通过使用整数指针的数组来说明使用指针数组的关键点。指针数组的例子也出现在下面几处：

- 3.3.5 节中我们使用了函数指针的数组；
- 6.1 节的“结构体的内存如何分配”中我们使用了结构体数组；
- 5.3.4 节中我们处理了 `argv` 数组。

本节的目的是说明这种方法的本质，来为接下来的几个例子打好基础。下面的代码片段声明一个整数指针的数组，为每个元素分配内存，然后把内存的内容初始化为元素的索引值：

```
int* arr[5];
for(int i=0; i<5; i++) {
    arr[i] = (int*)malloc(sizeof(int));
    *arr[i] = i;
}
```

如果把数组打印出来，得到的是数字 0 到 4。我们用 `arr[i]` 引用指针，用 `*arr[i]` 把值赋给指针引用的位置。别被数组表示法搞糊涂了，因为 `arr` 声明为指针数组，`arr[i]` 返回的是一个地址，当我们用 `*arr[i]` 解引指针时，得到是这个地址的内容。

我们也可以在循环体中使用下面这种等价的指针表示法：

```
*(arr+i) = (int*)malloc(sizeof(int));
**(arr+i) = i;
```

这种表示法更难理解，但是理解以后能加强你的 C 技能。在第二个语句中我们用了两层间接引用，掌握这种表示法将让你和初级 C 程序员有本质区别。

子表达式 `(arr+i)` 表示数组的第 `i` 个元素的地址，我们需要修改这个地址中的内容，所以用了子表达式 `*(arr+i)`。在第一条语句中我们将已分配的内存赋给这个位置。对 `(arr+i)` 子表达式做两次解引（如第二条语句所示），会返回所分配内存的位置，然后我们把 `i` 赋给它。图 4-9 说明了内存的分配情况。

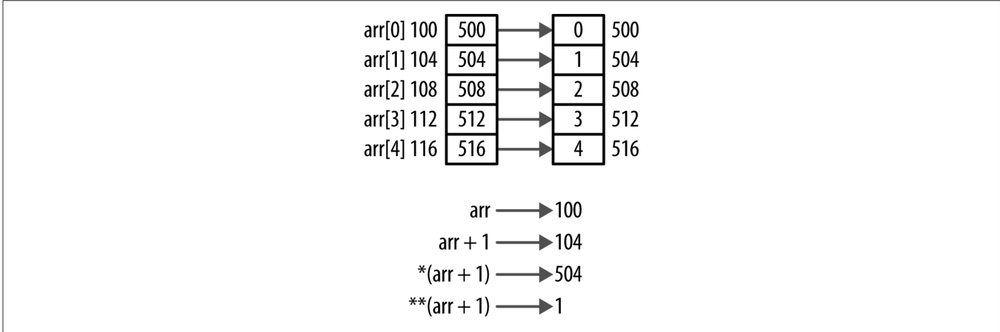


图 4-9：指针数组

比如说，`arr[1]` 位于地址 104，表达式 `(arr+1)` 为我们返回 104，用 `*(arr+1)` 则让我们得到其内容，在本例中，就是指针 504。再用 `**(arr+1)` 解引它就得到了 504 的内容，就是 1。

表 4-3 中列出了一些示例表达式。从左到右读指针表达式且不要忽略括号，这样会更容易理解其工作方式。

表4-3：指针数组表达式

表达式	值
`*arr[0]`	0
`**arr`	0
`**(arr+1)`	1
`arr[0][0]`	0
`arr[3][0]`	3

前三个表达式和前面解释的差不多，最后两个则有所不同。用指针的指针表示法能让我们知道正在处理的是指针数组，实际上我们的示例中也用到了。如果再看一下图 4-9，假设 `arr` 的每个元素指向一个长度为 1 的数组，那么最后两个表达式就能说通了，我们得到的是一个有 5 个元素的指针数组，这些指针指向一系列有 1 个元素的数组。

表达式 `arr[3][0]` 引用 `arr` 的第 4 个元素，然后是这个元素所指向的数组的第 1 个元素。表达式 `arr[3][1]` 有错误，因为第 4 个元素所指向的数组只有一个元素。

这个例子提醒我们可以创建不规则数组，这确实行得通，4.10 节讲的就是这个主题。

4.7　指针和多维数组

可以将多维数组的一部分看做子数组。比如说，二维数组的每一行都可以当做一维数组。这种行为会对我们用指针处理多维数组有所影响。

为了说明这种行为，我们创建一个二维数组并初始化，如下所示：

```
int matrix[2][5] = {{1,2,3,4,5},{6,7,8,9,10}};
```

然后打印元素的地址和值：

```
for(int i=0; i<2; i++) {
   for(int j=0; j<5; j++) {
      printf("matrix[%d][%d] Address: %p Value: %d\n",
                  i, j, &matrix[i][j], matrix[i][j]);
   }
}
```

输出如下所示：

```
matrix[0][0] Address: 100 Value: 1
matrix[0][1] Address: 104 Value: 2
matrix[0][2] Address: 108 Value: 3
matrix[0][3] Address: 112 Value: 4
matrix[0][4] Address: 116 Value: 5
matrix[1][0] Address: 120 Value: 6
matrix[1][1] Address: 124 Value: 7
matrix[1][2] Address: 128 Value: 8
matrix[1][3] Address: 132 Value: 9
matrix[1][4] Address: 136 Value: 10
```

数组按行－列顺序存储，也就是说，将第一行按顺序存入内存，后面紧接着第二行。内存分配如图 4-10 所示。

matrix[0][0] 100	1
matrix[0][1] 104	2
matrix[0][2] 108	3
matrix[0][3] 112	4
matrix[0][4] 116	5
matrix[1][0] 120	6
matrix[1][1] 124	7
matrix[1][2] 128	8
matrix[1][3] 132	9
matrix[1][4] 136	10

图 4-10：二维数组的内存分配

我们可以声明一个指针处理这个数组，如下所示：

```
int (*pmatrix)[5] = matrix;
```

(*pmatrix) 表达式声明了一个数组指针，上面的整条声明语句将 pmatrix 定义为一个指向二维数组的指针，该二维数组的元素类型是整数，每行有 5 个元素。如果我们把括号去掉就声明了 5 个元素的数组，数组元素的类型是整数指针。如果声明的列数不是 5，用该指针访问数组的结果则是不可预期的。

如果要用指针表示法访问第二个元素（就是 2），下面的代码看似合理：

```
printf("%p\n", matrix);
printf("%p\n", matrix + 1);
```

但输出却是：

```
100
120
```

matrix+1 返回的地址不是从数组开头偏移了 4，而是偏移了第一行的长度，20 字节。用 matrix 本身返回数组第一个元素的地址，二维数组是数组的数组，所以我们得到是一个拥有 5 个元素的整数数组的地址，它的长度是 20。我们可以用下面的语句验证这一点，它会打印出 20：

```
printf("%d\n",sizeof(matrix[0])); // 显示 20
```

要访问数组的第二个元素，需要给数组的第一行加上 1，像这样：*(matrix[0] + 1)。表达式 matrix[0] 返回数组第一行第一个元素的地址，这个地址是一个整数数组的地址，于是，给它加 1 实际加上的是一个整数的长度，得到的是第二个元素。输出结果是 104 和 2。

```
printf("%p %d\n", matrix[0] + 1, *(matrix[0] + 1));
```

我们可以用图文的形式来说明数组，如图 4-11 所示。

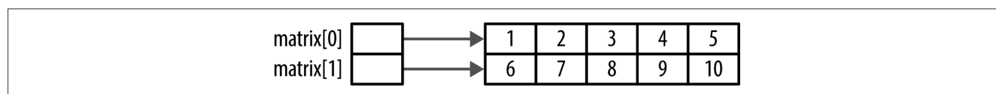


图 4-11：二维数组图示

图 4-12 可以解释二维数组表示法。

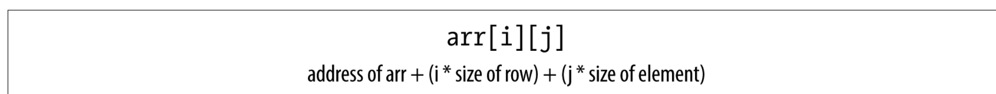


图 4-12：二维数组表示法

4.8 传递多维数组

给函数传递多维数组很容易让人迷惑，尤其是在用指针表示法的情况下。传递多维数组时，我们要决定在函数签名[1]中使用数组表示法还是指针表示法。还有一件要考虑的事情是如何传递数组的形态，这里所说的形态是指数组的维数及每一维的大小。要想在函数内部使用数组表示法，必须指定数组的形态，否则，编译器就无法使用下标。

要传递数组 matrix，可以这么写：

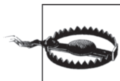
```
void display2DArray(int arr[][5], int rows) {
```

或者这么写：

```
void display2DArray(int (*arr)[5], int rows) {
```

这两种写法都指明了数组的列数，这很有必要，因为编译器需要知道每行有几个元素。如果没有传递这个信息，编译器就无法计算 4.7 节讲到的 arr[0][3] 这样的表达式。

在第一种写法中，表达式 arr[] 是数组指针的一个隐式声明，而第二种写法中的 (*arr) 表达式则是指针的一个显式声明。



下面的声明是错误的：

```
void display2DArray(int *arr[5], int rows) {
```

尽管不会产生语法错误，但是函数会认为传入的数组拥有 5 个整数指针。4.6 节讨论了指针数组。

这个函数的简单实现和调用方法如下：

```
void display2DArray(int arr[][5], int rows) {
    for (int i = 0; i<rows; i++) {
        for (int j = 0; j<5; j++) {
            printf("%d", arr[i][j]);
        }
        printf("\n");
    }
}

void main() {
    int matrix[2][5] = {
```

注 1：函数签名是指函数原型声明。——译者注

```
        {1, 2, 3, 4, 5},
        {6, 7, 8, 9, 10}
    };
    display2DArray(matrix, 2);
}
```

函数不会为这个数组分配内存，传递的只是地址。本次调用的程序栈状态如图 4-13 所示。

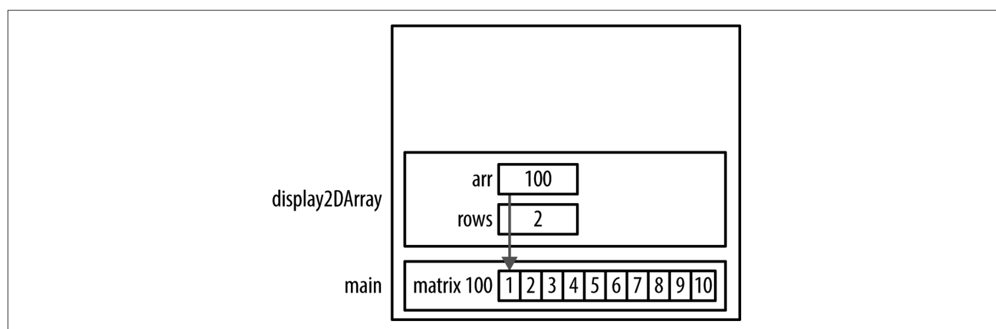


图 4-13：传递多维数组

你可能会遇到下面这样的函数，接受的参数是一个指针和行列数：

```
void display2DArrayUnknownSize(int *arr, int rows, int cols) {
    for(int i=0; i<rows; i++) {
        for(int j=0; j<cols; j++) {
            printf("%d ", *(arr + (i*cols) + j));
        }
        printf("\n");
    }
}
```

`printf` 语句通过给 `arr` 加上前面行的元素数（`i*cols`）以及表示当前列的 `j` 来计算每个元素的地址。要调用这个函数可以这么写：

```
display2DArrayUnknownSize(&matrix[0][0], 2, 5);
```

在函数内我们无法像下面这样使用数组下标：

```
printf("%d ", arr[i][j]);
```

原因是没有将指针声明为二维数组。不过，倒是可以像下面这样使用数组表示法。我们可以用一个下标，这样写只是解释为数组内部的偏移量，不能用两个下标是因为编译器不知道一维的长度：

```
printf("%d ", (arr+i)[j]);
```

这里传递的是 &matrix[0][0] 而不是 matrix，尽管用 matrix 也能运行，但是会产生编译警告，原因是指针类型不兼容。&matrix[0][0] 表达式是一个整数指针，而 matrix 则是一个整数数组的指针。

在传递二维以上的数组时，除了第一维以外，需要指定其他维度的长度。下面这个函数打印一个三维数组，声明中指定了数组的后二维。

```
void display3DArray(int (*arr)[2][4], int rows) {
    for(int i=0; i<rows; i++) {
        for(int j=0; j<2; j++) {
            printf("{");
            for(int k=0; k<4; k++) {
                printf("%d ", arr[i][j][k]);
            }
            printf("}");
        }
        printf("\n");
    }
}
```

下面说明如何调用这个函数：

```
int arr3d[3][2][4] = {
    {{1, 2, 3, 4}, {5, 6, 7, 8}},
    {{9, 10, 11, 12}, {13, 14, 15, 16}},
    {{17, 18, 19, 20}, {21, 22, 23, 24}}
};

display3DArray(arr3d,3);
```

输出如下所示：

```
{1 2 3 4 }{5 6 7 8 }
{9 10 11 12 }{13 14 15 16 }
{17 18 19 20 }{21 22 23 24 }
```

数组的内存分配如图 4-14 所示。

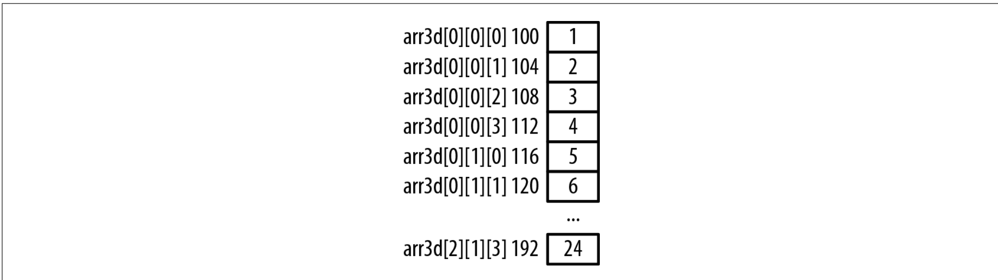


图 4-14：三维数组

arr3d[1] 表达式引用数组的第二行，是一个 2 行 4 列的二维数组的指针。arr3d[1][0] 引用数组的第二行第一列，是一个长度为 4 的一维数组的指针。

4.9 动态分配二维数组

为二维数组动态分配内存涉及几个问题：

- 数组元素是否需要连续；
- 数组是否规则。

一个声明如下的二维数组所分配的内存是连续的：

```
int matrix[2][5] = {{1,2,3,4,5},{6,7,8,9,10}};
```

不过，当我们用 malloc 这样的函数创建二维数组时，在内存分配上会有几种选择。由于我们可以将二维数组当做数组的数组，因而“内层”的数组没有理由一定要是连续的。如果对这种数组使用下标，数组的不连续对程序员是透明的。



连续性还会影响复制内存等其他操作，内存不连续就可能需要多次复制。

4.9.1 分配可能不连续的内存

下面的代码演示了如何创建一个内存可能不连续的二维数组。首先分配“外层”数组，然后分别用 malloc 语句为每一行分配。

```
int rows = 2;
int columns = 5;

int **matrix = (int **) malloc(rows * sizeof(int *));

for (int i = 0; i < rows; i++) {
    matrix[i] = (int *) malloc(columns * sizeof(int));
}
```

因为分别用了 malloc，所以内存不一定是连续的，如图 4-15 所示。

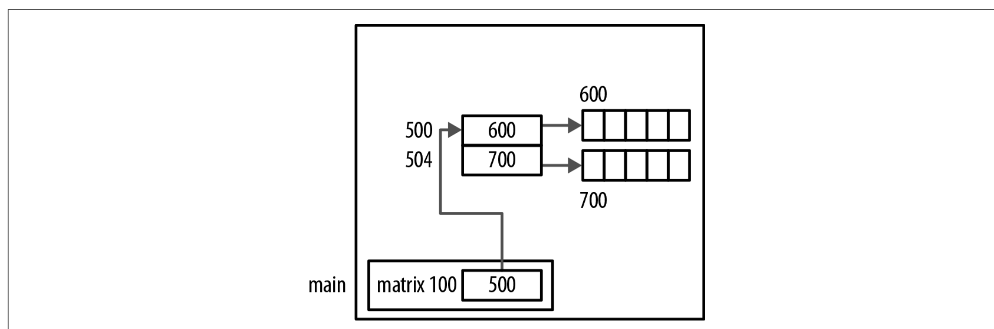


图 4-15：不连续分配

实际的分配情况取决于堆管理器和堆的状态，也有可能是连续的。

4.9.2　分配连续内存

我们会展示为二维数组分配连续内存的两种方法。第一种首先分配“外层”数组，然后是各行所需的所有内存。第二种一次性分配所有内存。

下面的代码片段演示了第一种技术，第一个 `malloc` 分配了一个整数指针的数组，一个元素用来存储一行的指针，这就是图 4-16 中在地址 500 处分配的内存块。第二个 `malloc` 在地址 600 处为所有的元素分配内存。在 `for` 循环中，我们将第二个 `malloc` 所分配的内存的一部分赋值给第一个数组的每个元素。

```
int rows = 2;
int columns = 5;
int **matrix = (int **) malloc(rows * sizeof(int *));
matrix[0] = (int *) malloc(rows * columns * sizeof(int));
for (int i = 1; i < rows; i++)
    matrix[i] = matrix[0] + i * columns;
```

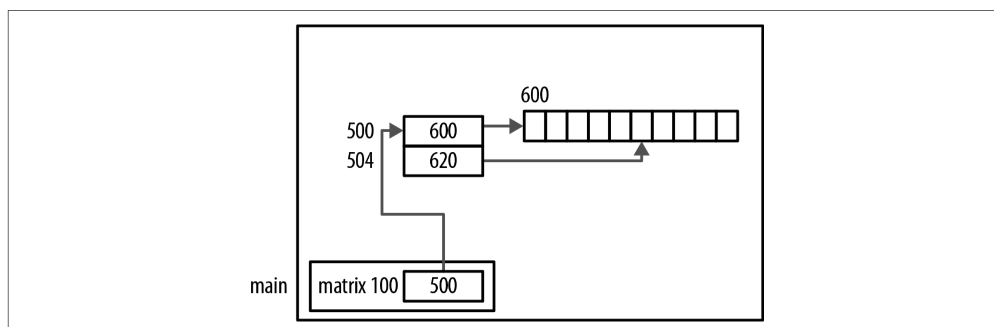


图 4-16：用两次 `malloc` 调用分配连续内存

从技术上讲，第一个数组的内存可以和数组“体”的内存分开，为数组“体”分配的内存是连续的。

下面是第二种技术，数组所需的所有内存是一次性分配的：

```
int *matrix = (int *)malloc(rows * columns * sizeof(int));
```

分配的情况如图 4-17 所示。

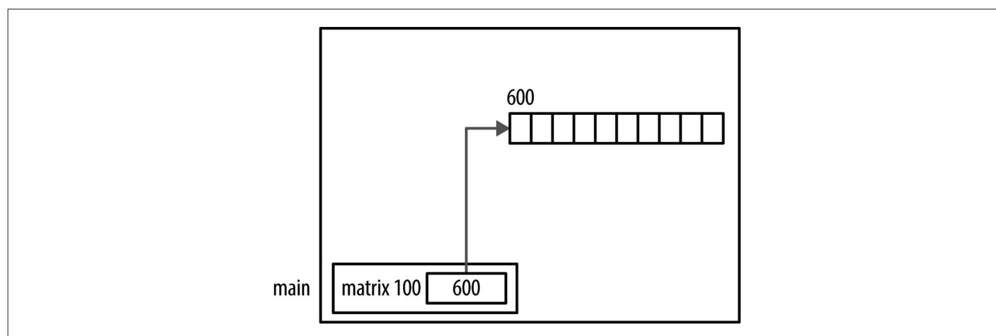


图 4-17：用一次 malloc 调用分配连续内存

后面的代码用到这个数组时不能使用下标，必须手动计算索引，如下代码片段所示。每个元素被初始化为其索引的积：

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < columns; j++) {
        *(matrix + (i*columns) + j) = i*j;
    }
}
```

不能使用数组下标是因为我们丢失了允许编译器使用下标所需的“形态”信息。这个概念在 4.8 节讲过了。

实际项目中很少使用这种方法，但它确实说明了二维数组概念和内存的一维本质的关系。便捷的二维数组表示法让这种映射变得透明且更容易使用。

我们已经演示了为二维数组分配连续内存的两种方法，具体使用哪种要看应用程序的需要。不过第二种方法是为“整个”数组分配一块内存。

4.10　不规则数组和指针

不规则数组是每一行的列数不一样的二维数组，其原理如图 4-18 所示，图中的数组有 3 行，每行有不同的列数。

		行		
	0	1	2	3
0	0	1	2	3
列 1	4	5		
2	6	7	8	

图 4-18：不规则数组

在了解如何创建不规则数组之前，让我们先看一下用复合字面量创建的二维数组。复合字面量是一种 C 构造，前面看起来像类型转换操作，后面跟着花括号括起来的初始化列表。下面是整数常量和整数数组的例子，我们将其作为声明的一部分：

```
(const int) {100}
(int[3]) {10, 20, 30}
```

下面的声明把数组声明为整数指针的数组，然后用复合字面量语句块进行初始化，由此创建了数组 `arr1`。

```
int (*(arr1[])) = {
    (int[]) {0, 1, 2},
    (int[]) {3, 4, 5},
    (int[]) {6, 7, 8}};
```

这个数组有 3 行 3 列，将数组元素用数字 0 到 8 按行 – 列顺序初始化。图 4-19 说明了数组的内存布局。

arr1[0][0]	100	0
arr1[0][1]	104	1
arr1[0][2]	108	2
arr1[1][0]	112	3
arr1[1][1]	116	4
arr1[1][2]	120	5
arr1[2][0]	124	6
arr1[2][1]	128	7
arr1[2][2]	132	8

图 4-19：二维数组

下面的代码片段打印每个数组元素的地址和值：

```
for(int j=0; j<3; j++) {
    for(int i=0; i<3; i++) {
        printf("arr1[%d][%d] Address: %p Value: %d\n",
                j, i, &arr1[j][i], arr1[j][i]);
    }
    printf("\n");
}
```

执行后会得到如下输出：

```
arr1[0][0] Address: 0x100 Value: 0
arr1[0][1] Address: 0x104 Value: 1
arr1[0][2] Address: 0x108 Value: 2

arr1[1][0] Address: 0x112 Value: 3
arr1[1][1] Address: 0x116 Value: 4
arr1[1][2] Address: 0x120 Value: 5

arr1[2][0] Address: 0x124 Value: 6
arr1[2][1] Address: 0x128 Value: 7
arr1[2][2] Address: 0x132 Value: 8
```

稍微修改一下声明就可以得到一个不规则数组，就是图 4-18 中展示的那个。数组声明如下：

```
int (*(arr2[])) = {
    (int[]) {0, 1, 2, 3},
    (int[]) {4, 5},
    (int[]) {6, 7, 8}};
```

我们用了 3 个复合字面量声明不规则数组，然后从 0 开始按行－列顺序初始化数组元素。下面的代码片段会打印数组来验证创建是否正确，因为每行的列数不同，所以需要 3 个 `for` 循环：

```
int row = 0;
for(int i=0; i<4; i++) {
    printf("layer1[%d][%d] Address: %p Value: %d\n",
            row, i, &arr2[row][i], arr2[row][i]);
}
printf("\n");

row = 1;
for(int i=0; i<2; i++) {
    printf("layer1[%d][%d] Address: %p Value: %d\n",
            row, i, &arr2[row][i], arr2[row][i]);
}
printf("\n");

row = 2;
for(int i=0; i<3; i++) {
    printf("layer1[%d][%d] Address: %p Value: %d\n",
            row, i, &arr2[row][i], arr2[row][i]);
}
printf("\n");
```

输出如下：

```
arr2[0][0] Address: 0x000100 Value: 0
arr2[0][1] Address: 0x000104 Value: 1
```

```
arr2[0][2] Address: 0x000108 Value: 2
arr2[0][3] Address: 0x000112 Value: 3

arr2[1][0] Address: 0x000116 Value: 4
arr2[1][1] Address: 0x000120 Value: 5

arr2[2][0] Address: 0x000124 Value: 6
arr2[2][1] Address: 0x000128 Value: 7
arr2[2][2] Address: 0x000132 Value: 8
```

图 4-20 说明了这个数组的内存布局。

arr2[0][0] 100	0
arr2[0][1] 104	1
arr2[0][2] 108	2
arr2[0][3] 112	3
arr2[1][0] 116	4
arr2[1][1] 120	5
arr2[2][0] 124	6
arr2[2][1] 128	7
arr2[2][2] 132	8

图 4-20：不规则数组的内存分配

在这些例子中，我们访问数组内容时用的是数组表示法而不是指针表示法，这样更易读，也好理解。不过，也可以用指针表示法。

复合字面量在创建不规则数组时很有用，不过访问不规则数组的元素比较别扭，上面的例子就用了 3 个 `for` 循环。如果有一个单独的数组来维护每行的长度，那么这个例子就可以简化。你可以在 C 中创建不规则数组，不过要考虑好它能起的作用是否值得花费相应的精力。

4.11　小结

本章首先简述了数组，然后研究了数组表示法和指针表示法的异同。我们可以使用 `malloc` 类函数创建数组，这类函数提供了比传统数组声明更大的灵活性。我们也学习了如何用 `realloc` 函数调整数组的内存。

为数组动态分配内存可能会比较难，对于二维或多维数组，我们得小心确保数组分配在连续的内存上。

我们也探索了在传递和返回数组时可能产生的问题，通常需要给函数传递数组长度以便函数能正确处理数组。我们还研究了如何在 C 中创建不规则数组。

第5章 指针和字符串

字符串可以分配在内存的不同区域，通常使用指针来支持字符串操作。指针支持动态分配字符串和将字符串作为参数传递给函数。深入理解指针及指针与字符串结合的用法可以让程序员开发出有效而且高效的应用程序。

字符串是很多应用程序的常见组成部分，也是一个复杂的主题。在本章中，我们会探索声明和初始化字符串的不同方法，研究 C 程序中字面量池的使用及其影响。此外，我们还会看到比较、复制和拼接字符串等常见字符串操作。

字符串通常以字符指针的形式传递给函数和从函数返回。我们可以用字符指针传递字符串，也可以用字符常量的指针传递字符串，后者可以避免字符串被函数修改。本章用到的很多例子能更好地说明第 3 章中提到的原理，不同之处在于本章的例子无需将长度传递给函数。

我们也可以从函数返回字符串，从而满足某个请求。可以将这个字符串从外面传给函数并由函数修改，也可以在函数内部分配，还可以返回静态分配的字符串。本章会逐一探讨这些方法。

我们也会研究函数指针的用法以及如何用函数指针辅助排序操作。理解这些情况下指针如何工作是本章关注的重点。

5.1 字符串基础

字符串是以 ASCII 字符 `NUL` 结尾的字符序列。ASCII 字符 `NUL` 表示为 `\0`。字符串

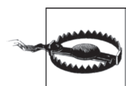
通常存储在数组或者从堆上分配的内存中。不过，并非所有的字符数组都是字符串，字符数组可能没有 NUL 字符。字符数组也用来表示布尔值等小的整数单元，以节省内存空间。

C 中有两种类型的字符串。

- 单字节字符串
  由 char 数据类型组成的序列。
- 宽字符串
  由 wchar_t 数据类型组成的序列。

wchar_t 数据类型用来表示宽字符，要么是 16 位宽，要么是 32 位宽。这两种字符串都以 NUL 结尾。可以在 string.h 中找到单字节字符串函数，而在 wchar.h 中找到宽字符串函数。除非特别指明，本章用到的都是单字节字符串。创建宽字符主要用来支持非拉丁字符集，对于支持外语的应用程序很有用。

字符串的长度是字符串中除了 NUL 字符之外的字符数。为字符串分配内存时，要记得为所有的字符再加上 NUL 字符分配足够的空间。



记住，NULL 和 NUL 不同。NULL 用来表示特殊的指针，通常定义为 ((void*)0)，而 NUL 是一个 char，定义为 \0，两者不能混用。

字符常量是单引号引起来的字符序列。字符常量通常由一个字符组成，也可以包含多个字符，比如转义字符。在 C 中，它们的类型是 int，如下所示：

```
printf("%d\n",sizeof(char));
printf("%d\n",sizeof('a'));
```

执行上述代码可以看到 char 的长度是 1，而字符字面量的长度是 4。这个看似异常的现象乃语言设计者有意为之。

5.1.1　字符串声明

声明字符串的方式有三种：字面量、字符数组和字符指针。字符串字面量是用双引号引起来的字符序列，常用来进行初始化，它们位于字符串字面量池中，我们会在下一节讨论。

不要把字符串字面量和单引号引起来的字符搞混——后者是字符字面量。在后面的各节我们会看到，把字符字面量当做字符串字面量用会出问题。

下面是一个字符数组的例子，我们声明了一个 header 数组，最多可以持有 31 个字符。因为字符串需要以 NUL 结尾，所以如果我们声明一个数组拥有 32 个字符，那么只能用 31 个元素来保存实际字符串的文本。字符串在内存中的位置取决于声明的位置，我们会在 5.1.3 节中探究这个问题。

```
char header[32];
```

字符指针如下所示，由于没有初始化，也就没有引用字符串，当前还没有指定字符串的长度和位置。

```
char *header;
```

5.1.2 字符串字面量池

定义字面量时通常会将其分配在字面量池中，这个内存区域保存了组成字符串的字符序列。多次用到同一个字面量时，字面量池中通常只有一份副本。这样会减少应用程序占用的内存。通常认为字面量是不可变的，因此只有一份副本不会有什么问题。不过，认定只有一份副本或者字面量不可变不是一种好做法，大部分编译器有关闭字面量池的选项，一旦关闭，字面量可能生成多个副本，每个副本拥有自己的地址。



GCC 用 -fwritable-strings 选项来关闭字符串池。在 Microsoft Visual Studio 中，/GF 选项会打开字符串池。

图 5-1 说明了字面量池的内存分配方式。

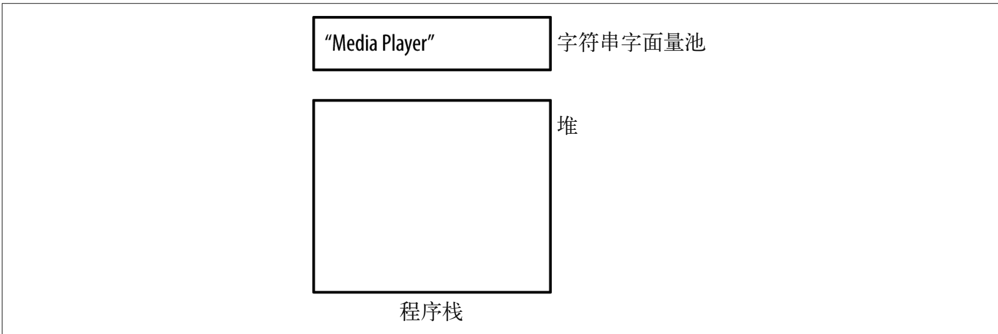


图 5-1：字符串字面量池

字符串字面量一般分配在只读内存中，所以是不可变的。字符串字面量在哪里使用，或者它是全局、静态或局部的都无关紧要，从这个角度讲，字符串字面量不存在作用域的概念。

字符串字面量不是常量的情况

在大部分编译器中，我们将字符串字面量看做常量，无法修改字符串。不过，在有些编译器中（比如 GCC），字符串字面量是可修改的。看下面这个例子：

```
char *tabHeader = "Sound";
*tabHeader = 'L';
printf("%s\n",tabHeader); // 打印 "Lound"
```

这样会把字面量改成 "Lound"，这通常不是我们期望的结果，因此应该避免这么做。像下面这样把变量声明为常量可以解决一部分问题。任何修改字符串的尝试都会造成编译时错误：

```
const char *tabHeader = "Sound";
```

5.1.3　字符串初始化

初始化字符串采用的方法取决于变量是被声明为字符数组还是字符指针，字符串所用的内存要么是数组要么是指针指向的一块内存。我们可以用字符串字面量或者一系列字符初始化字符串，或者从别的地方（比如说标准输入）得到字符。接下来我们会研究这些方法。

1. 初始化char数组

我们可以用初始化操作符初始化 char 数组。在下例中，header 数组被初始化为字符串字面量中所包含的字符：

```
char header[] = "Media Player";
```

字面量 "Media Player" 的长度为 12 个字符，表示这个字面量需要 13 字节，我们就为数组分配了 13 字节来持有字符串。初始化操作会把这些字符复制到数组中，以 NUL 结尾，如图 5-2 所示，这里假设在 main 函数中声明数组。

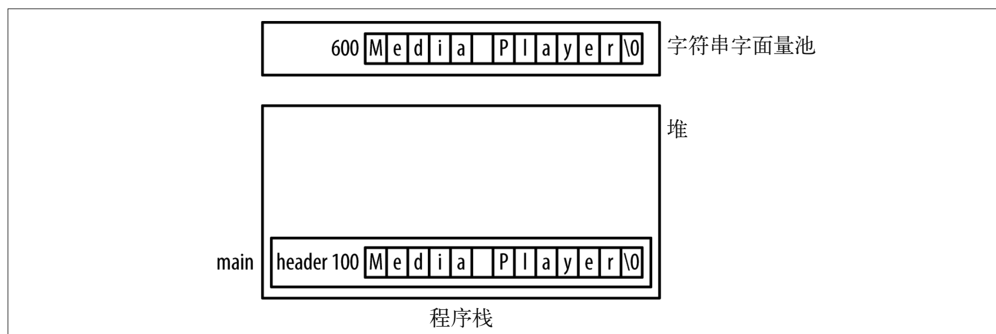


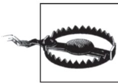
图 5-2：初始化 char 数组

我们也可以用 strcpy 函数初始化数组，5.2.2 节会详细讨论 strcpy。下面的代码片段将字符串字面量复制到了数组中。

```
char header[13];
strcpy(header,"Media Player");
```

更笨的办法是把字符逐个赋给数组元素，如下所示：

```
header[0] = 'M';
header[1] = 'e';
     ...
header[12] = '\0';
```



下面的赋值是不合法的，我们不能把字符串字面量的地址赋给数组名字。

```
char header2[];
header2 = "Media Player";
```

2. 初始化char指针

动态内存分配可以提供更多的灵活性，当然也可能会让内存存在得更久。下面的声明用来说明这种技术：

```
char *header;
```

初始化这个字符串的常见方法是使用 malloc 和 strcpy 函数分配内存并将字面量复制到字符串中，如下所示：

```
char *header = (char*) malloc(strlen("Media Player")+1);
strcpy(header,"Media Player");
```

假设这段代码在 main 函数中，图 5-3 显示了程序栈的状态。

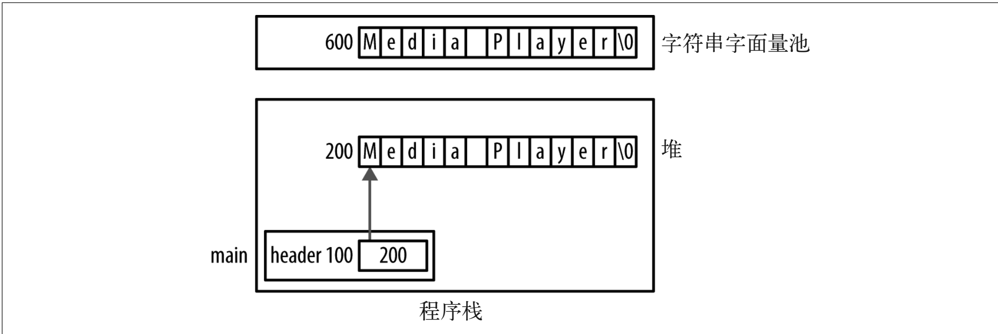
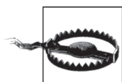


图 5-3：初始化 char 指针

前面用到 malloc 函数的地方，我们对字符串字面量使用了 strlen 函数，也可以如下所示明确指定长度：

```
char *header = (char*) malloc(13);
```



在决定 malloc 函数要用到的字符串长度时，要注意以下事项。

- 一定要记得算上终结符 NUL。
- 不要用 sizeof 操作符，而是用 strlen 函数来确定已有字符串的长度。sizeof 操作符会返回数组和指针的长度，而不是字符串的长度。

如果不用字符串字面量和 strcpy 函数初始化字符串，我们也可以这么做：

```
*(header + 0) = 'M';
*(header + 1) = 'e';
    ...
*(header + 12) = '\0';
```

我们可以将字符串字面量的地址直接赋给字符指针，如下所示。不过，这样不会产生字符串的副本，如图 5-4 所示。

```
char *header = "Media Player";
```

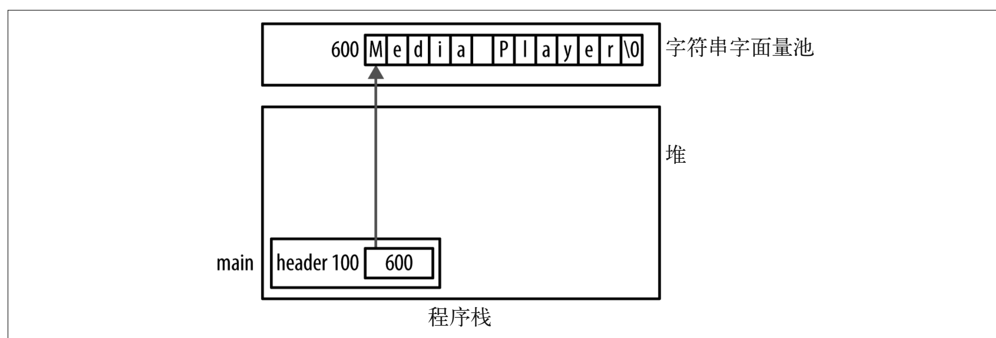
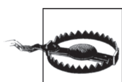


图 5-4：复制字符串字面量的地址到指针中



试图用字符字面量来初始化 char 指针不会起作用。因为字符字面量是 int 类型，这其实是尝试把整数赋给字符指针。这样经常会造成应用程序在解引指针时终止：

```
char* prefix = '+'; // 不合法
```

正确的做法是像下面这样用 malloc 函数：

```
prefix = (char*)malloc(2);
*prefix = '+';
*(prefix+1) = 0;
```

3. 从标准输入初始化字符串

也可以用标准输入等外部源初始化字符串。不过，在从标准输入读入字符串时可能

会出错，下面是个例子。这里会出问题是因为我们在使用 command 变量之前没有为其分配内存：

```
char *command;
printf("Enter a Command: ");
scanf("%s",command);
```

要解决这个问题需要首先为指针分配内存，或者用定长数组代替指针。不过，用户输入的数据可能比我们所能装下的要多，第 4 章讨论了更健壮的方法。

4. 字符串位置小结

我们可能将字符串分配在几个地方，下例解释了几种可能的变化，图 5-5 说明了这些字符串在内存中的布局。

```
char* globalHeader = "Chapter";
char globalArrayHeader[] = "Chapter";

void displayHeader() {
    static char* staticHeader = "Chapter";
    char* localHeader = "Chapter";
    static char staticArrayHeader[] = "Chapter";
    char localArrayHeader[] = "Chapter";
    char* heapHeader = (char*)malloc(strlen("Chapter")+1);
    strcpy(heapHeader,"Chapter");
}
```

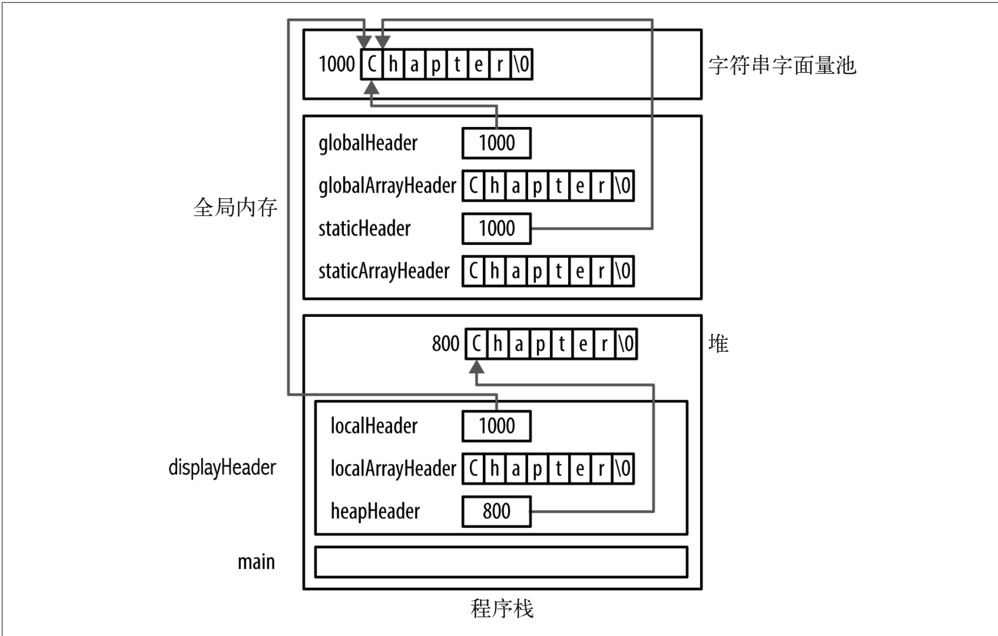


图 5-5：字符串的内存分配

知道字符串存储的位置对理解程序的工作原理以及用指针访问字符串有帮助。字符串的位置决定它能存在多久，以及程序的哪些部分可以访问它。比如说，分配在全局内存的字符串会一直存在，也可以被多个函数访问；静态字符串也一直存在，不过只有定义它们的函数才能访问，分配在堆上的内存在释放之前会一直存在，也可以被多个函数访问。理解这些东西能让你作出更好的选择。

5.2 标准字符串操作

在本节中，我们会研究指针在常见字符串操作中的使用，包括比较、复制和拼接。

5.2.1 比较字符串

字符串比较是应用程序不可分割的一部分，我们会深入研究如何比较字符串，因为不正确的比较会产生误导或无效结果，理解字符串的比较能帮助你避开不正确的操作。这种认识能让你触类旁通。

比较字符串的标准方法是用 `strcmp` 函数，原型如下：

```
int strcmp(const char *s1, const char *s2);
```

要比较的两个字符串都以指向 `char` 常量的指针的形式传递，这让我们可以放心地使用这个函数，而不用担心传入的字符串被修改。这个函数返回以下三种值之一。

- 负数
  如果按字典序（字母序）`s1` 比 `s2` 小就返回负数。

- 0
  如果两个字符串相等就返回 0。

- 正数
  如果按字典序 `s1` 比 `s2` 大就返回正数。

正数和负数返回值对于按字母序对字符串进行排序很有用，使用这个函数判断相等性的用法如下所示。用户的输入存储在 `command` 中，然后跟字符串字面量比较：

```
char command[16];

printf("Enter a Command: ");
scanf("%s", command);
if (strcmp(command, "Quit") == 0) {
    printf("The command was Quit");
} else {
    printf("The command was not Quit");
}
```

本例的内存分配见图 5-6。

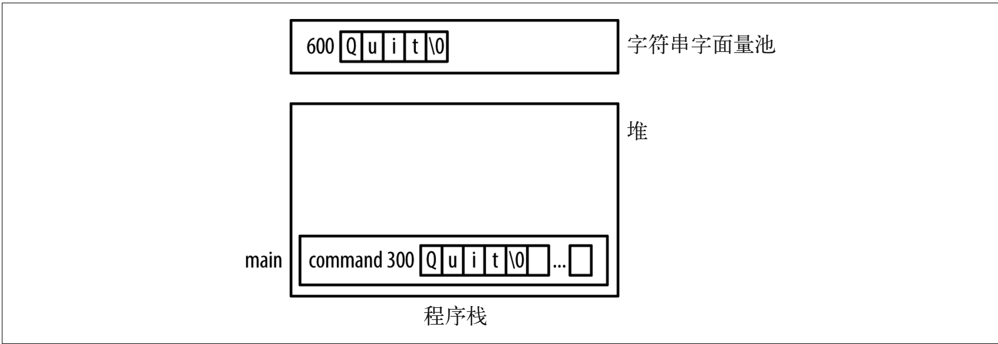


图 5-6：strcmp 示例

比较两个字符串有几种不正确的写法，第一种试图用赋值操作符作比较，如下：

```
char command[16];

printf("Enter a Command: ");
scanf("%s",command);
if(command = "Quit") {
   ...
```

首先，这不是作比较，其次，这样会导致类型不兼容的语法错误，我们不能把字符串字面量地址赋给数组名字。在本例中，我们试图把字符串字面量的地址（也就是 600）赋给 command。command 是数组，不用数组下标就把一个值赋给这个变量是不可能的。

另一种方法是用相等操作符：

```
char command[16];

printf("Enter a Command: ");
scanf("%s",command);
if(command == "Quit") {
   ...
```

这样会得到假，因为我们比较的是 command 的地址（300）和字符串字面量的地址（600）。相等操作符比较的是地址，而不是地址中的内容，用数组名字或者字符串字面量就会返回地址。

5.2.2 复制字符串

复制字符串是常见的操作，通常用 strcpy 函数实现，其原型如下：

```
char* strcpy(char *s1, const char *s2);
```

本节会讲到基本的复制过程和常见的陷阱。假设要将一个已有的字符串复制到动态分配的缓冲区中（也可以用字符数组）。

有一类常见的应用程序会读入一系列字符串，挨个存入占据最少内存的数组。要实现这一点，可以创建一个长度足以容纳用户可能输入的最长字符串的数组，并且把字符串读入这个数组。有了读入的字符串，我们就能分配合适的内存。基本的方法是这样的：

(1) 用一个很长的 char 数组读入字符串；
(2) 用 malloc 分配恰好容纳字符串的适量内存；
(3) 用 strcpy 把字符串复制到动态分配的内存中。

下面的代码说明了这种技术。names 数组会持有每个读入的名字的指针，而 count 变量则指定下一个可用的数组元素。name 数组用来持有读入的字符串，每个读入的名字都可以重复利用它，malloc 函数分配每个字符串所需的内存并将其赋给 names 中下一个可用的元素。之后将名字复制到新分配的内存中：

```
char name[32];
char *names[30];
size_t count = 0;

printf("Enter a name: ");
scanf("%s",name);
names[count] = (char*)malloc(strlen(name)+1);
strcpy(names[count],name);
count++;
```

我们可以在一个循环中重复这个操作，每次迭代增加 count 值。图 5-7 说明了对于读入的名字 Sam，这些处理的内存布局。

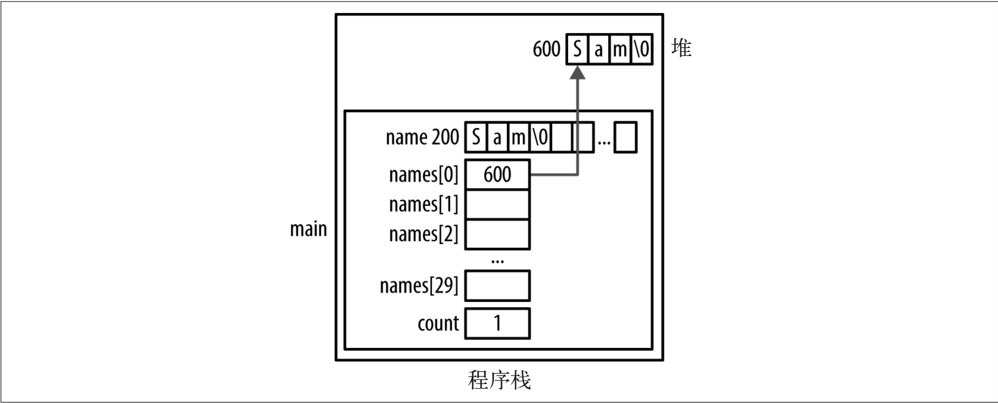


图 5-7：复制字符串

两个指针可以引用同一个字符串。两个指针引用同一个地址称为*别名*，这个话题会在第 8 章讲到。尽管通常情况下这不是问题，但要知道，把一个指针赋值给另一个指针不会复制字符串，只是复制了字符串的地址。

为了说明这一点，下面声明了页眉指针的数组。我们将字符串字面量的地址赋给了索引为 12 的页面，接着，把 `pageHeaders[12]` 中的指针复制到 `pageHeaders[13]`。现在这两个指针都指向同一个字符串字面量。这里复制的是指针而不是字符串：

```
char *pageHeaders[300];

pageHeaders[12] = "Amorphous Compounds";
pageHeaders[13] = pageHeaders[12];
```

图 5-8 解释了这些赋值操作。

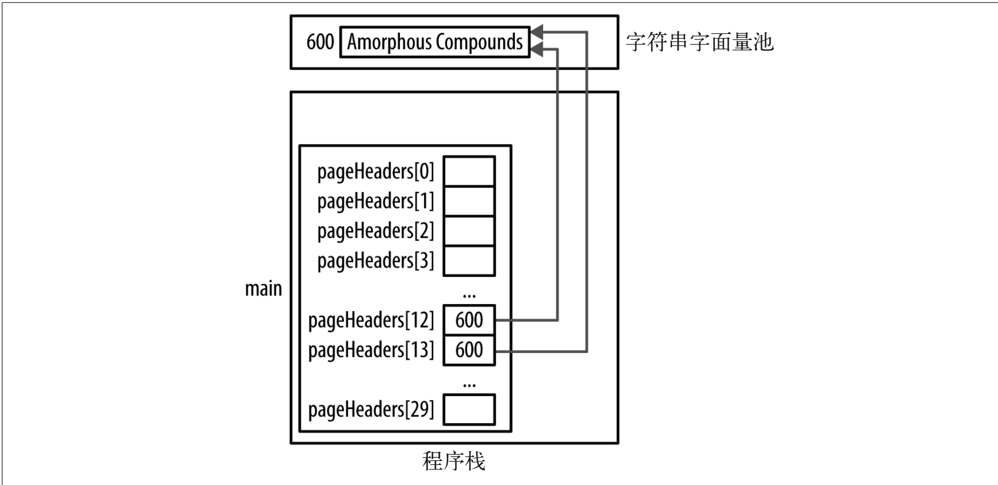


图 5-8：复制指针的效果

5.2.3　拼接字符串

字符串拼接涉及两个字符串的合并。`strcat` 函数经常用来执行这种操作，这个函数接受两个字符串指针作为参数，然后把两者拼接起来并返回拼接结果的指针。这个函数的原型如下：

```
char *strcat(char *s1, const char *s2);
```

此函数把第二个字符串拼接到第一个的结尾，第二个字符串是以常量 `char` 指针的形式传递的。函数不会分配内存，这意味着第一个字符串必须足够长，能容纳拼接

后的结果，否则函数可能会越界写入，导致不可预期的行为。函数的返回值的地址跟第一个参数的地址一样。这在某些情况下比较方便，比如这个函数作为 printf 函数的参数时。

为了说明这个函数的用法，我们会组合两个错误消息字符串。第一个是前缀，第二个是具体的错误消息。如下所示，我们首先在缓冲区中为两个字符串分配足够的内存，然后把第一个字符串复制到缓冲区，最后将第二个字符串和缓冲区拼接：

```
char* error = "ERROR: ";
char* errorMessage = "Not enough memory";

char* buffer = (char*)malloc(strlen(error)+strlen(errorMessage)+1);
strcpy(buffer,error);
strcat(buffer, errorMessage);

printf("%s\n", buffer);
printf("%s\n", error);
printf("%s\n", errorMessage);
```

我们给 malloc 函数的参数加 1 是为了容纳 NUL 字符。假设第一个字面量在内存中的位置就在第二个字面量前面，这段代码的输出会像下面这样。图 5-9 说明了内存分配情况。

```
ERROR: Not enough memory
ERROR:
Not enough memory
```

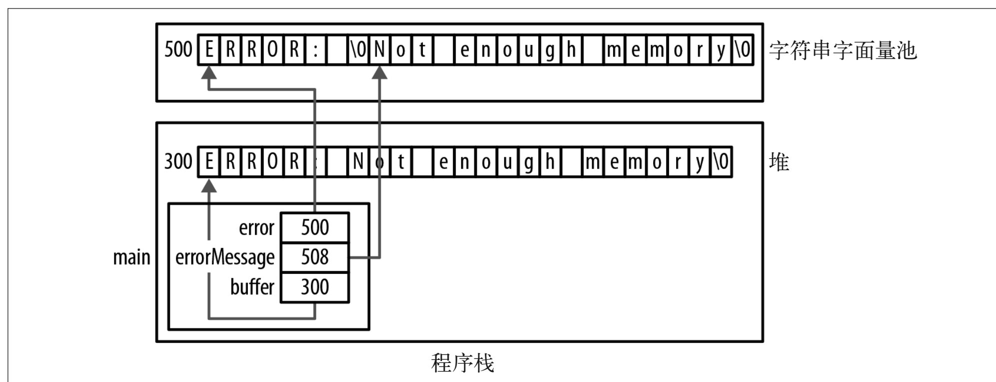


图 5-9：正确的拼接操作

如果我们没有为拼接后的字符串分配独立的内存，就可能会覆写第二个字符串，下面这个没有用到缓冲区的例子会说明这一点。我们仍然假设第一个字面量在内存中的位置就在第二个字面量前面：

```
char* error = "ERROR: ";
char* errorMessage = "Not enough memory";

strcat(error, errorMessage);
printf("%s\n", error);
printf("%s\n", errorMessage);
```

这段代码的输出如下：

```
ERROR: Not enough memory
ot enough memory
```

errorMessage 字符串会左移一个字符，原因是拼接后的结果覆写了 errorMessage。字面量 "Not enough memory" 紧跟在第一个字面量之后，因此覆写了第二个字面量。图 5-10 解释了这一点，字面量池的状态显示在左边，右边是复制操作后的状态。

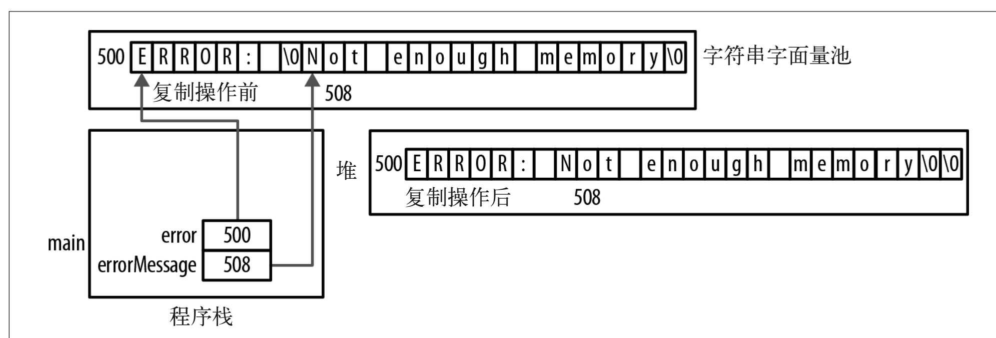


图 5-10：不正确的字符串拼接操作

如果我们像下面这样用 char 数组而不是用指针来存储字符串，就不一定能工作了：

```
char error[] = "ERROR: ";
char errorMessage[] = "Not enough memory";
```

如果用下面这个 strcpy 调用会得到一个语法错误，这是因为我们试图把函数返回的指针赋给数组名字，这类操作不合法：

```
error = strcat(error, errorMessage);
```

如果像下面这样去掉赋值，就可能会有内存访问的漏洞，因为复制操作会覆写栈帧的一部分。这里假设在函数内部声明数组，如图 5-11 所示。无论源字符串是存储在字符串字面量池中还是栈帧中，都不应该用来直接存放拼接后的结果，一定要专门为拼接结果分配内存：

```
strcat(error, errorMessage);
```

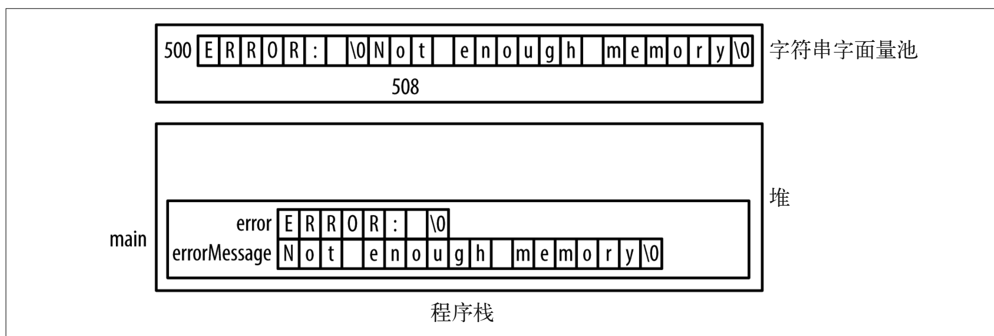


图 5-11：覆写栈帧

拼接字符串时容易犯错的另一个地方是使用字符字面量而不是字符串字面量。在下例中，我们将一个字符串拼接到一个路径字符串后，这样是能如期工作的：

```
char* path = "C:";
char* currentPath = (char*) malloc(strlen(path)+2);
currentPath = strcat(currentPath,"\\");
```

因为额外的字符和 NUL 字符需要空间，我们在 malloc 调用中给字符串长度加了 2。因为在字符串字面量中用了转义序列，所以这里拼接的是一个反斜杠字符。

不过，如果使用字符字面量，如下所示，那么就会得到一个运行时错误，原因是第二个参数被错误地解释为 char 类型变量的地址 [1]：

```
currentPath = strcat(path,'\\');
```

5.3　传递字符串

传递字符串很简单，在函数调用中，用一个计算结果是 char 类型变量地址的表达式即可。在参数列表中，把参数声明为 char 指针。有趣的事情发生在函数内部使用字符串时。我们首先会在 5.3.1 节和 5.3.2 节研究如何传递简单字符串，然后在 5.3.3 节中研究如何传递需要初始化的字符串。把字符串作为参数传递给应用程序会在 5.3.4 节中讲解。

5.3.1　传递简单字符串

取决于不同的字符串声明方式，有几种方法可以把字符串的地址传递给函数。在本节中，我们会利用一个模拟 strlen 的函数说明这些技术，该函数的实现如下代码

注 1：此处其实是个整数，而参数是 char*，所以整数被当成了地址。——译者注

所示。我们用括号来强制后面的自增操作符先执行，使得指针加 1。否则加 1 的就是 string 引用的字符了，这不是我们想要的结果。

```
size_t stringLength(char* string) {
    size_t length = 0;
    while(*(string++)) {
        length++;
    }
    return length;
}
```



字符串实际上应该以 char 常量的指针的形式传递，5.3.2 节会讨论这一点。

让我们从下面的声明开始：

```
char simpleArray[] = "simple string";
char *simplePtr = (char*)malloc(strlen("simple string")+1);
strcpy(simplePtr, "simple string");
```

要对这个指针调用此函数，只要用指针名字即可：

```
printf("%d\n",stringLength(simplePtr));
```

要使用数组调用函数，我们有三种选择，如下所示。在第一个语句中，我们用了数组的名字，这会返回其地址。在第二个语句中，显式使用了取地址操作符，不过这样写有冗余，没有必要，而且会产生警告。在第三个语句中，我们对数组第一个元素用了取地址操作符，这样可以工作，不过有点繁琐：

```
printf("%d\n",stringLength(simpleArray));
printf("%d\n",stringLength(&simpleArray));
printf("%d\n",stringLength(&simpleArray[0]));
```

图 5-12 说明了 stringLength 函数的内存分配情况。

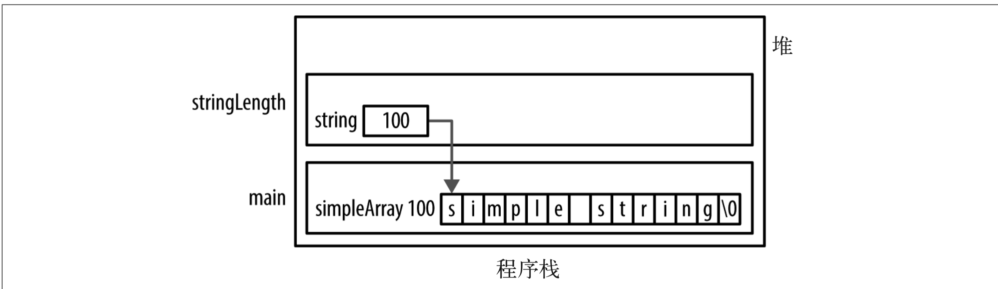


图 5-12：传递字符串

现在让我们把注意力转移到形参的声明方式上。在前面 stringLength 的实现中，我们把参数声明为 char 指针，不过也可以像下面这样用数组表示法：

```
size_t stringLength(char string[]) { ... }
```

函数体还是一样，这个变化不会对函数的调用方式及其行为造成影响。

5.3.2　传递字符常量的指针

以字符常量指针的形式传递字符串指针是很常见也很有用的技术，这样可以用指针传递字符串，同时也能防止传递的字符串被修改。下面对 5.3.1 节中的 stringLength 函数更好的实现就是利用了这种声明：

```
size_t stringLength(const char* string) {
    size_t length = 0;
    while(*(string++)) {
        length++;
    }
    return length;
}
```

如果我们试图像下面这样修改原字符串，那么就会产生一个编译时错误消息：

```
size_t stringLength(const char* string) {
      ...
    *string = 'A';
      ...
}
```

5.3.3　传递需要初始化的字符串

有些情况下我们想让函数返回一个由该函数初始化的字符串。假设我们想传递一个部件的信息，比如名字和数量，然后让函数返回表示这个信息的格式化字符串。通过把格式化处理放在函数内部，我们可以在程序的不同部分重用这个函数。

不过，我们得决定是给函数传递一个空缓冲区让它填充并返回，还是让函数动态分配缓冲区并返回。

要传递缓冲区：

- 必须传递缓冲区的地址和长度；
- 调用者负责释放缓冲区；
- 函数通常返回缓冲区的指针。

这种方法把分配和释放缓冲区的责任都交给了调用者。虽然没有必要，返回缓冲区

指针很常见，strcpy 或类似函数就是这种情况。下面的 format 函数说明了这种方法：

```
char* format(char *buffer, size_t size,
        const char* name, size_t quantity, size_t weight) {
    snprintf(buffer, size, "Item: %s Quantity: %u Weight: %u",
            name, quantity, weight);
    return buffer;
}
```

这里用了 snprintf 函数来简化字符串格式化，该函数写入第一个参数指向的缓冲区。第二个参数指定缓冲区的长度，函数不会越过缓冲区写入。其他方面，这个函数和 printf 函数的行为一样。

下面的语句说明了这个函数的用法。它假设缓冲区已被声明为一个数组。如果已动态分配缓冲区的内存，则需要传入分配的内存大小，而不是使用函数的大小。

```
printf("%s\n",format(buffer,sizeof(buffer),"Axle",25,45));
```

输出如下：

```
Item: Axle Quantity: 25 Weight: 45
```

通过返回缓冲区的指针，我们可以将函数作为 printf 函数的参数。

还有一种方法是传递 NULL 作为缓冲区地址，这表示调用者不想提供缓冲区，或者它不确定缓冲区应该是多大。这样的函数实现列在了下面，在计算长度时，10 + 10 子表达式表示数量和重量可能的最大宽度，而 1 则是为 NUL 终结符留下空间：

```
char* format(char *buffer, size_t size,
        const char* name, size_t quantity, size_t weight) {

    char *formatString = "Item: %s Quantity: %u Weight: %u";
    size_t formatStringLength = strlen(formatString)-6;
    size_t nameLength = strlen(name);
    size_t length = formatStringLength + nameLength +
            10 + 10 + 1;

    if(buffer == NULL) {
        buffer = (char*)malloc(length);
        size = length;
    }
    snprintf(buffer, size, formatString, name, quantity, weight);
    return buffer;
}
```

函数使用的变量取决于应用程序的需要。第二种方法的主要缺点在于调用者现在要负责释放分配的内存，调用者需要对函数的使用方法了如指掌，否则可能很容易产生内存泄漏。

5.3.4 给应用程序传递参数

main 函数通常是应用程序第一个执行的函数。对基于命令行的程序来说，通过为其传递信息来打开某种行为的开关或控制某种行为很常见。可以用这些参数来指定要处理的文件或是配置应用程序的输出。比如说，Linux 的 ls 命令会基于接收到的参数列出当前目录下的文件。

C 用传统的 argc 和 argv 参数支持命令行参数。第一个参数 argc，是一个指定传递的参数数量的整数。系统至少会传递一个参数，这个参数是可执行文件的名字。第二个参数 argv，通常被看做字符串指针的一维数组，每个指针引用一个命令行参数。

下面的 main 函数只是简单地列出了它的参数，每行一个。在这个版本中，argv 被声明为一个 char 指针的指针。

```
int main(int argc, char** argv) {
    for(int i=0; i<argc; i++) {
        printf("argv[%d] %s\n",i,argv[i]);
    }
    ...
}
```

程序可以用下面的命令执行：

```
process.exe -f names.txt limit=12 -verbose
```

输出如下：

```
argv[0] c:/process.exe
argv[1] -f
argv[2] names.txt
argv[3] limit=12
argv[4] -verbose
```

使用空格将每个命令行参数分开，这个程序的内存分配如图 5-13 所示。

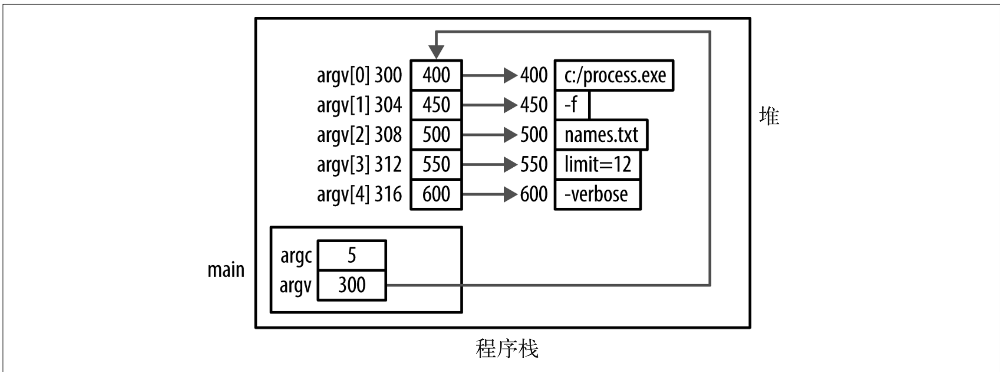


图 5-13：使用 argc/argv

argv 的声明可以简化如下：

```
int main(int argc, char* argv[]) {
```

这跟 char** argv 是等价的，1.4.1 节详细解释了这种表示法。

5.4 返回字符串

函数返回字符串时，它返回的实际是字符串的地址。这里应该关注的主要问题是如何返回合法的地址，要做到这一点，可以返回以下三种对象之一的引用：

- 字面量；
- 动态分配的内存；
- 本地字符串变量。

5.4.1 返回字面量的地址

返回字面量的例子如下所示，利用一个整数码从四个处理中心选择一个。这个函数的目的是把处理中心的名字作为字符串返回。在本例中，它只是返回了字面量的地址：

```
char* returnALiteral(int code) {
    switch(code) {
        case 100:
            return "Boston Processing Center";
        case 200:
            return "Denver Processing Center";
        case 300:
            return "Atlanta Processing Center";
        case 400:
            return "San Jose Processing Center";
    }
}
```

这段代码会工作得很好。唯一需要记住的一点是我们并非总是将字符串字面量看做常量，5.1.2 节讨论过这一点。也可以像下例这样声明静态字面量，我们增加了 subCode 字段来选择不同的中心，这么做的好处是无需在不同的地方使用同一个字面量，也就不会因为打错字而引入错误了：

```
char* returnAStaticLiteral(int code, int subCode) {
    static char* bpCenter = "Boston Processing Center";
    static char* dpCenter = "Denver Processing Center";
    static char* apCenter = "Atlanta Processing Center";
    static char* sjpCenter = "San Jose Processing Center";

    switch(code) {
        case 100:
```

```
            return bpCenter;
        case 135:
            if(subCode <35) {
                return dpCenter;
            } else {
                return bpCenter;
            }
        case 200:
            return dpCenter;
        case 300:
            return apCenter;
        case 400:
            return sjpCenter;
    }
}
```

针对多个不同目的返回同一个静态字符串的指针可能会有问题。考虑下面的函数，这是在 5.3.3 节中开发的 `format` 函数的变体。将一个部件的信息传递给函数，然后返回一个表示这个部件的格式化字符串：

```
char* staticFormat(const char* name, size_t quantity, size_t weight) {
    static char buffer[64]; // 假设缓冲区足够大
    sprintf(buffer, "Item: %s Quantity: %u Weight: %u",
            name, quantity, weight);
    return buffer;
}
```

为缓冲区分配 64 字节可能够，也可能不够，就本例的目的而言，我们会忽略这个潜在的问题。这种方法的主要问题用如下代码片段说明：

```
char* part1 = staticFormat("Axle",25,45);
char* part2 = staticFormat("Piston",55,5);
printf("%s\n",part1);
printf("%s\n",part2);
```

执行后得到如下输出：

```
Item: Piston Quantity: 55 Weight: 5
Item: Piston Quantity: 55 Weight: 5
```

`staticFormat` 两次调用都使用同一个静态缓冲区，后一次调用会覆写前一次调用的结果。

5.4.2 返回动态分配内存的地址

如果需要从函数返回字符串，我们可以在堆上分配字符串的内存然后返回其地址。我们会开发一个 `blanks` 函数来说明这种技术，这个函数会返回一个包含一系列代表“制表符”的空白的字符串，如下所示。函数接受一个指定制表符序列长度的整

数参数：

```
char* blanks(int number) {
    char* spaces = (char*) malloc(number + 1);
    int i;
    for (i = 0; i<number; i++) {
        spaces[i] = ' ';
    }
    spaces[number] = '\0';
    return spaces;
}

    ...
    char *tmp = blanks(5);
```

将 NUL 终结符赋给由 number 索引的数组的最后一个元素，图 5-14 说明了本例的内存分配，它显示了 blanks 函数返回前后应用程序的状态。

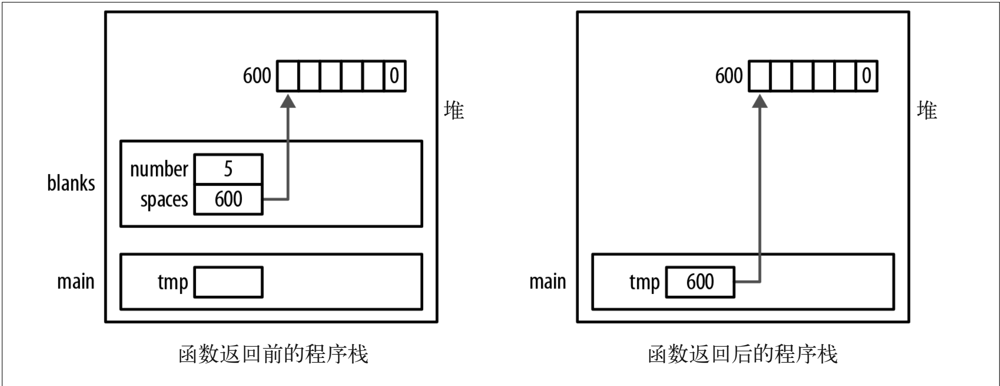


图 5-14：返回动态分配的字符串

释放返回的内存是函数调用者的责任，如果不再需要内存但没有将其释放会造成内存泄漏。下面是一个内存泄漏的例子，printf 函数中使用了字符串，但是接着它的地址就丢失了，因为我们没有保存：

```
printf("[%s]\n",blanks(5));
```

一个更安全的方法如下所示：

```
char *tmp = blanks(5);
printf("[%s]\n",tmp);
free(tmp);
```

返回局部字符串的地址

返回局部字符串的地址可能会有问题，如果内存被别的栈帧覆写就会损坏，应该避

免使用这种方法，这里作解释只是为了说明实际使用这种方法的潜在问题。

我们重写前面的 blanks 函数，如下所示。在函数内部声明一个数组，而不是动态分配内存，这个数组位于栈帧上。函数返回数组的地址：

```
#define MAX_TAB_LENGTH 32

char* blanks(int number) {
    char spaces[MAX_TAB_LENGTH];
    int i;
    for (i = 0; i < number && i < MAX_TAB_LENGTH; i++) {
        spaces[i] = ' ';
    }
    spaces[i] = '\0';
    return spaces;
}
```

执行函数后会返回数组的地址，但是之后下一次函数调用会覆写这块内存区域。解引指针后该内存地址的内容可能已经改变。图 5-15 说明了程序栈的状态。

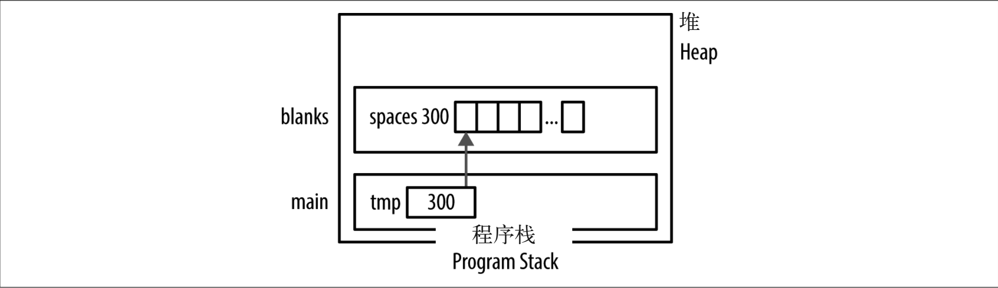


图 5-15：返回局部字符串的地址

5.5 函数指针和字符串

我们已经在 3.3 节中深入讨论过函数指针了，它们是控制程序执行的一种非常灵活的方法。在本节中，我们会通过将比较函数传递给排序函数来说明这种能力。排序函数通过比较数组的元素来判断是否交换数组元素，比较决定了数组是按升序还是降序（或者其他排序策略）排列。通过传递一个函数来控制比较，排序函数会变得更灵活。传递不同的比较函数可以让同一个排序函数以不同的方式工作。

我们使用的比较函数根据数组的元素大小写决定排序顺序。下面的 compare 和 compareIgnoreCase 会根据大小写比较字符串。在用 strcmp 函数比较字符串之前，compareIgnoreCase 函数会先把字符串转换成小写。5.2.1 节中已经讨论过 strcmp 函数了。stringToLower 函数返回动态分配内存的指针，这意味着一旦不

需要就应该将其释放掉。

```
int compare(const char* s1, const char* s2) {
    return strcmp(s1,s2);
}

int compareIgnoreCase(const char* s1, const char* s2) {
    char* t1 = stringToLower(s1);
    char* t2 = stringToLower(s2);
    int result = strcmp(t1, t2);
    free(t1);
    free(t2);
    return result;
}
```

`stringToLower` 函数如下所示，它将传递进来的字符串用小写的形式返回：

```
char* stringToLower(const char* string) {
    char *tmp = (char*) malloc(strlen(string) + 1);
    char *start = tmp;
    while (*string != 0) {
        *tmp++ = tolower(*string++);
    }
    *tmp = 0;
    return start;
}
```

使用如下的类型定义声明我们要使用的函数指针：

```
typedef int (fptrOperation)(const char*, const char*);
```

下面的 `sort` 函数的实现基于冒泡排序算法，我们将数组地址、数组长度以及一个控制排序的函数指针传递给它。在 `if` 语句中，调用传递进来的函数并传递数组的两个元素，它会判断这两个元素是否需要交换。

```
void sort(char *array[], int size, fptrOperation operation) {
    int swap = 1;
    while(swap) {
        swap = 0;
        for(int i=0; i<size-1; i++) {
            if(operation(array[i],array[i+1]) > 0){
                swap = 1;
                char *tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
            }
        }
    }
}
```

打印函数会显示数组的内容：

```
void displayNames(char* names[], int size) {
    for(int i=0; i<size; i++) {
        printf("%s   ",names[i]);
    }
    printf("\n");
}
```

我们可以用两个比较函数中的任意一个作为参数调用 `sort` 函数。下面用 `compare` 函数进行区分大小写的排序：

```
char* names[] = {"Bob", "Ted", "Carol", "Alice", "alice"};
sort(names,5,compare);
displayNames(names,5);
```

输出如下：

```
Alice  Bob  Carol  Ted  alice
```

如果使用 `compareIgnoreCase` 函数，输出则是这样：

```
Alice  alice  Bob  Carol  Ted
```

这样 `sort` 函数就灵活得多了，我们可以设计并传递自己想要的任意简单或复杂的操作来控制排序，而不需要针对不同的排序需求写不同的排序函数。

5.6　小结

本章重点讲解了字符串操作和指针的使用，字符串的结构和在内存中的位置会影响其使用。指针提供了操作字符串的灵活工具，但是也可能造成误用字符串。

我们也提到了字符串字面量和字面量池的用法，理解字面量有助于我们理解某些字符串赋值操作没有按照预期工作的原因，这跟字符串的初始化密切相关，这一点我们也进行了深入讨论。我们还研究了一些标准的字符串操作，指出了哪些地方容易出问题。

给函数传递字符串和从函数返回字符串是常见的操作，我们也详细讨论了关于这类操作的潜在问题，包括返回局部字符串时容易出现的问题。此外，我们还讨论了字符常量指针的用法。

最后，我们用函数指针说明了实现排序函数的强大方法，这种方法不局限于排序函数，也可以应用到其他领域。

第 6 章

指针和结构体

我们可以使用 C 的结构体来表示数据结构元素，比如链表或树的节点，指针是把这些元素联系到一起的纽带。理解指针对常见数据结构多种功能的支持可以为创建数据结构提供便利。在本章中，我们会探索 C 中结构体内存分配的基础和几种常见数据结构的实现。

结构体加强了数组等集合的实用性。要创建实体的数组（比如有多个字段的颜色类型），如果不用结构体的话，就得为每个字段声明一个数组，然后把每个字段的值放在每个数组的同一个索引下。不过，有了结构体，我们可以只声明一个数组，其中的每个元素是一个结构体的实例。

本章继续拓展前面所学的指针概念，包括结构体的数组表示法、结构体的内存分配、结构体内存管理技术以及函数指针的用法。

我们会从结构体的内存分配开始，理解内存分配可以解释很多操作的工作原理。接着我们会介绍减少堆管理开销的技术。

最后一节说明如何用指针创建一系列数据结构。首先是链表，链表是其他几种数据结构的基础，最后是树数据结构，它没有用到链表。

6.1　介绍

声明 C 结构体的方式有多种。本节只看其中两种，因为我们主要关注的是结构体和指针的配合使用。在第一种方法中，我们用 `struct` 关键字声明一个结构体。在第

二种方法中，我们使用类型定义。在下面的声明中，结构体的名字前面加了下划线，这不是必需的，不过通常作为命名约定。_person 结构体包括了名字、职位和年龄三个字段。

```
struct _person {
    char* firstName;
    char* lastName;
    char* title;
    unsigned int age;
};
```

结构体的声明经常使用 typedef 关键字简化之后的使用。下面说明如何对 _person 结构体用 typedef 关键字：

```
typedef struct _person {
    char* firstName;
    char* lastName;
    char* title;
    unsigned int age;
} Person;
```

person 的实例声明如下：

```
Person person;
```

我们也可以声明一个 Person 指针并为它分配内存，如下所示：

```
Person *ptrPerson;
ptrPerson = (Person*) malloc(sizeof(Person));
```

如果使用结构体的简单声明（像 person 那样），那么就用点表示法来访问其字段。在下例中，我们给 firstName 和 age 字段赋了值：

```
Person person;
person.firstName = (char*)malloc(strlen("Emily")+1);
strcpy(person.firstName,"Emily");
person.age = 23;
```

不过，如果使用结构体指针，就需要用箭头操作符，如下所示。这个操作符由一个横线和一个大于号组成：

```
Person *ptrPerson;
ptrPerson = (Person*)malloc(sizeof(Person));
ptrPerson->firstName = (char*)malloc(strlen("Emily")+1);
strcpy(ptrPerson->firstName,"Emily");
ptrPerson->age = 23;
```

我们不一定非得用箭头操作符，可以先解引指针然后用点操作符，如下所示，我们

又执行了一遍赋值操作：

```
Person *ptrPerson;
ptrPerson = (Person*)malloc(sizeof(Person));
(*ptrPerson).firstName = (char*)malloc(strlen("Emily")+1);
strcpy((*ptrPerson).firstName,"Emily");
(*ptrPerson).age = 23;
```

这种方法有些笨拙，不过你偶尔还能看到有人使用它。

为结构体分配内存

为结构体分配内存时，分配的内存大小至少是各个字段的长度和。不过，实际长度通常会大于这个和，因为结构体的各字段之间可能会有填充。某些数据类型需要对齐到特定边界就会产生填充。比如说，短整数通常对齐到能被 2 整除的地址上，而整数对齐到能被 4 整除的地址上。

这些额外内存的分配意味着几个问题：

- 要谨慎使用指针算术运算；
- 结构体数组的元素之间可能存在额外的内存。

比如说，如果为上一节中出现的 Person 结构体的实例分配内存，会分配 16 字节——每个元素 4 字节。下面这个版本的 Person 用短整数来代替无符号整数作为 age 的类型。这样分配的内存大小还是一样，因为结构体末尾填充了 2 字节：

```
typedef struct _alternatePerson {
    char* firstName;
    char* lastName;
    char* title;
    short age;
} AlternatePerson;
```

在下面的代码片段中，我们声明了 Person 和 AlternatePerson 结构体的实例，然后打印结构体的长度。它们的长度相同，都是 16 字节：

```
Person person;
AlternatePerson otherPerson;

printf("%d\n",sizeof(Person));             // 打印 16
printf("%d\n",sizeof(AlternatePerson));    // 打印 16
```

如果我们创建一个 AlternatePerson 的数组（如下所示），那么每个数组元素之间会有填充，如图 6-1 所示。阴影区域表示数组元素之间的空隙。

```
AlternatePerson people[30];
```

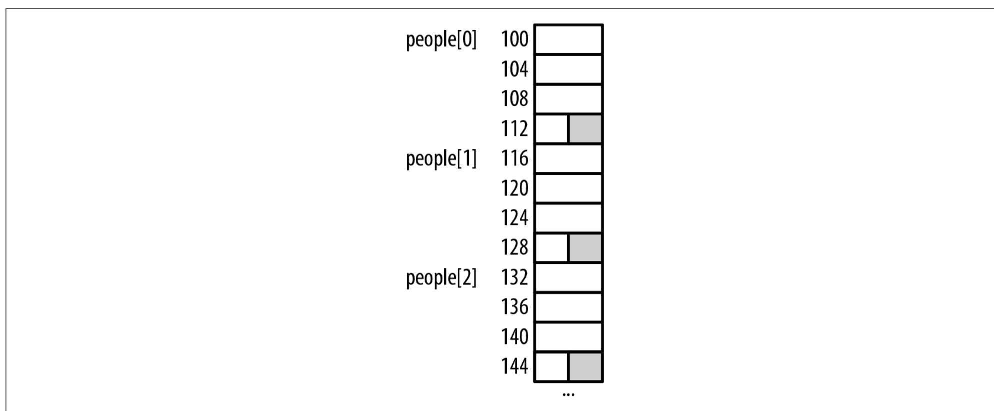


图 6-1：AlternatePerson 的数组

如果我们把 age 字段移到结构体的两个字段中间，那么空隙就处于结构体内部。根据访问结构体的方式，这可能会很重要。

6.2 结构体释放问题

在为结构体分配内存时，运行时系统不会自动为结构体内部的指针分配内存。类似地，当结构体消失时，运行时系统也不会自动释放结构体内部的指针指向的内存。

考虑如下结构体：

```
typedef struct _person {
    char* firstName;
    char* lastName;
    char* title;
    uint age;
} Person;
```

当我们声明这个类型的变量或者为这个类型动态分配内存时，三个指针会包含垃圾数据。在下面的代码片段中，我们声明了 Person，其内存分配如图 6-2 所示，三个点表示未初始化的内存。

```
void processPerson() {
   Person person;
   ...
}
```

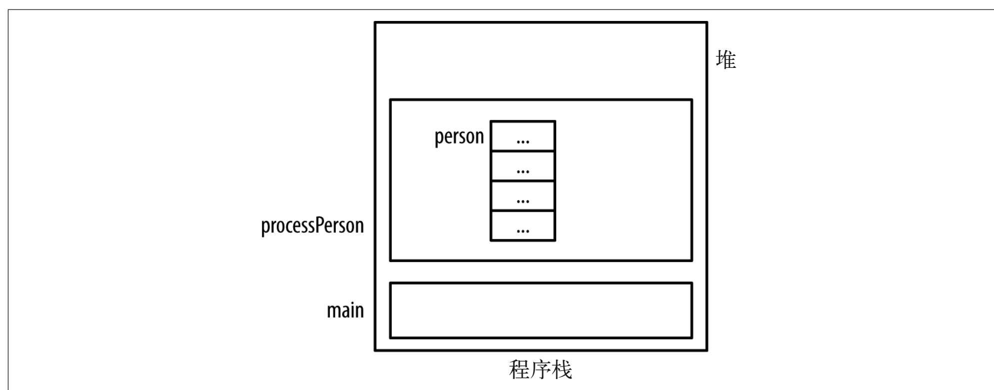


图 6-2：未初始化的 Person 结构体

在这个结构体的初始化阶段，会为每个字段赋一个值。对于指针字段，我们会从堆上分配内存并把地址赋给每个指针：

```
void initializePerson(Person *person, const char* fn,
        const char* ln, const char* title, uint age) {
    person->firstName = (char*) malloc(strlen(fn) + 1);
    strcpy(person->firstName, fn);
    person->lastName = (char*) malloc(strlen(ln) + 1);
    strcpy(person->lastName, ln);
    person->title = (char*) malloc(strlen(title) + 1);
    strcpy(person->title, title);
    person->age = age;
}
```

可以如下这样使用这个函数，图 6-3 说明了内存分配情况：

```
void processPerson() {
   Person person;
   initializePerson(&person, "Peter", "Underwood", "Manager", 36);
   ...
}
int main() {
   processPerson();
   ...
}
```

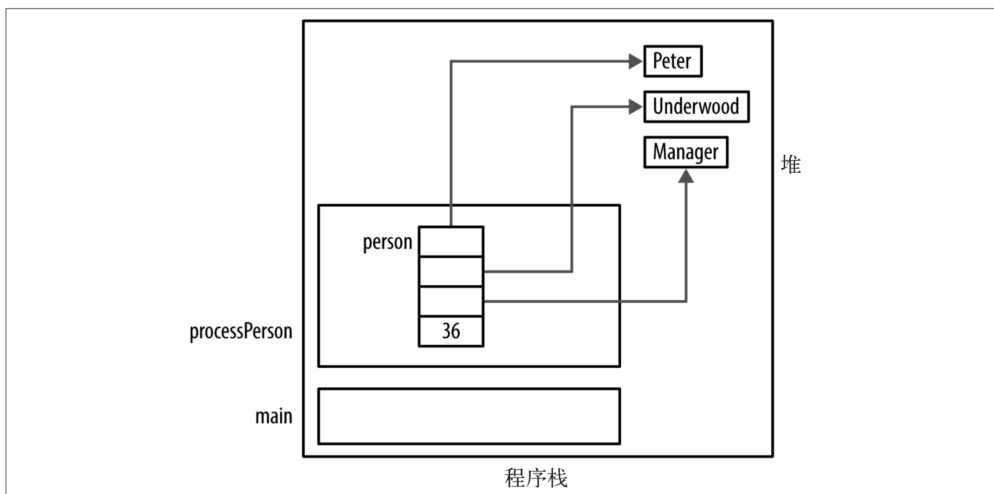


图 6-3：初始化的 Person 结构体

因为这个声明是函数的一部分，函数返回后 person 的内存会消失。不过，动态分配的内存不会被释放，仍然保存在堆上。不幸的是，我们丢失了它们的地址，因此无法将其释放，从而导致了内存泄漏。

用完这个实例后需要释放内存。下面的函数会释放之前创建实例时分配的内存：

```
void deallocatePerson(Person *person) {
    free(person->firstName);
    free(person->lastName);
    free(person->title);
}
```

我们需要在函数结束前调用这个函数：

```
void processPerson() {
   Person person;
   initializePerson(&person, "Peter", "Underwood", "Manager", 36);
    ...
   deallocatePerson(&person);
}
```

另外，我们必需记得调用 initialize 和 deallocate 函数，但诸如 C++ 这类面向对象的编程语言会自动为对象调用这些操作。

如果用 Person 指针，必须释放如下所示的 person：

```
void processPerson() {
    Person *ptrPerson;
    ptrPerson = (Person*) malloc(sizeof(Person));
```

```
        initializePerson(ptrPerson, "Peter", "Underwood", "Manager", 36);
          ...
        deallocatePerson(ptrPerson);
        free(ptrPerson);
    }
```

图 6-4 说明了内存分配情况。

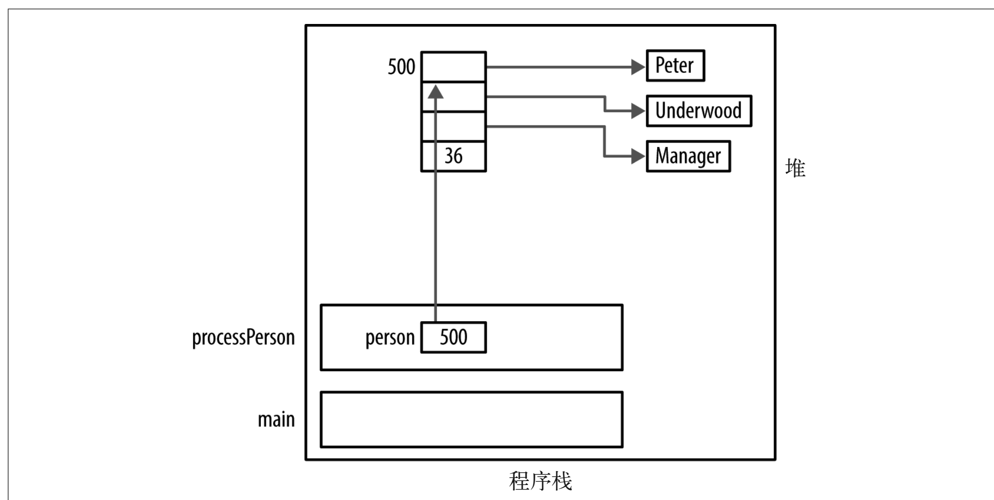


图 6-4：指向 person 实例的指针

6.3 避免malloc/free开销

重复分配然后释放结构体会产生一些开销，可能导致巨大的性能瓶颈。解决这个问题的一种办法是为分配的结构体单独维护一个表。当用户不再需要某个结构体实例时，将其返回结构体池中。当我们需要某个实例时，从结构体池中获取一个对象。如果池中没有可用的元素，我们就动态分配一个实例。这种方法高效地维护一个结构体池，能按需使用和重复使用内存。

为了说明这种方法，我们会用之前定义的 Person 结构体。用数组维护结构体池，也可以用链表等更复杂的表，6.4.1 节有相关说明。为了让示例简单，我们用了指针数组，声明如下：

```
#define LIST_SIZE 10
Person *list[LIST_SIZE];
```

使用表之前需要先初始化。下面的函数为数组每个元素赋值 NULL：

```
void initializeList() {
    for(int i=0; i<LIST_SIZE; i++) {
```

```
        list[i] = NULL;
    }
}
```

我们用两个函数来添加和获取结构体。第一个是 getPerson 函数，如下所示。如果存在可用的结构体，这个函数从表中获取一个。将数组的元素跟 NULL 比较，返回第一个非空的元素，然后将它在 list 中的位置赋值为 NULL。如果没有可用的结构体，那就创建并返回一个新的 Person 实例。这样就避免了每次需要结构体时都动态分配内存的开销，我们只在池中为空时才分配内存。返回实例的初始化可以在返回之前就做好，也可以由调用者来做，取决于应用程序的需要。

```
Person *getPerson() {
    for(int i=0; i<LIST_SIZE; i++) {
        if(list[i] != NULL) {
            Person *ptr = list[i];
            list[i] = NULL;
            return ptr;
        }
    }
    Person *person = (Person*)malloc(sizeof(Person));
    return person;
}
```

第二个函数是 returnPerson，这个函数要么将结构体返回表，要么把结构体释放掉。我们会检查数组元素看看有没有 NULL 值，有的话就将 person 添加到那个位置，然后返回指针。如果表满了，就用 deallocatePerson 函数释放 person 内的指针，然后释放 person，最后返回 NULL。

```
Person *returnPerson(Person *person) {
    for(int i=0; i<LIST_SIZE; i++) {
        if(list[i] == NULL) {
            list[i] = person;
            return person;
        }
    }
    deallocatePerson(person);
    free(person);
    return NULL;
}
```

下面的代码说明了表的初始化，以及如何将一个结构体添加到表中：

```
initializeList();
Person *ptrPerson;

ptrPerson = getPerson();
initializePerson(ptrPerson,"Ralph","Fitsgerald","Mr.",35);
displayPerson(*ptrPerson);
returnPerson(ptrPerson);
```

这种方法有个问题，就是表的长度。如果表太短，那么就需要更频繁地分配并释放内存。如果表太长而没有使用结构体，那么就会浪费大量的内存，也不能用在其他地方。可以用更复杂的表管理策略来管理表的长度。

6.4　用指针支持数据结构

指针可以为简单或复杂的数据结构提供更多的灵活性。这些灵活性可能来自动态内存分配，也可能来自切换指针引用的便利性。内存无需像数组那样是连续的，只要总的内存大小对就可以。

在本节中，我们会研究几种可以用指针实现的常用数据结构。很多 C 库都会提供这里提到的数据结构，而且会提供更广泛的支持。不过，理解如何实现这些数据结构对于实现非标准的数据结构很有帮助。在某些平台上可能无法使用这些库，开发者需要实现自己的版本。

我们会研究以下四种不同的数据结构。

- 链表
  单链表
- 队列
  简单的先进先出队列
- 栈
  简单的栈
- 树
  二叉树

我们会结合函数指针和这些数据结构来说明它们处理通用结构时的强大能力。链表是非常有用的数据结构，我们会把它作为实现队列和栈的基础。

我们会利用一个雇员结构体说明这些数据结构。比如说，链表由互相连接的节点组成，每个节点会持有用户提供的数据，雇员结构体如下所示。`unsigned char` 数据类型用来表示年龄，它足够装下人类的年龄了：

```
typedef struct _employee{
    char name[32];
    unsigned char age;
} Employee;
```

每个名字用一个数组表示，对于这个字段，char 指针可能是更灵活的数据类型，不过简单起见，我们还是选择了 char 数组。

我们会开发两个比较函数，第一个比较两个雇员然后返回一个整数，它模仿了 strcmp 函数，返回值 0 表示两个雇员结构体相等，–1 表示第一个雇员比第二个雇员小，1 表示第一个雇员比第二个雇员大。第二个函数则用来打印雇员：

```
int compareEmployee(Employee *e1, Employee *e2) {
    return strcmp(e1->name, e2->name);
}

void displayEmployee(Employee* employee) {
    printf("%s\t%d\n", employee->name, employee->age);
}
```

此外，我们还会用到两个函数指针，定义如下。DISPLAY 函数指针表示一个接受 void* 参数并返回 void 的函数，目的是显示数据。第二个函数指针 COMPARE 比较两个指针引用的数据，它的返回值是 0、–1 或 1，我们在介绍 compareEmployee 函数的时候解释过了：

```
typedef void(*DISPLAY)(void*);
typedef int(*COMPARE)(void*, void*);
```

6.4.1 单链表

链表是由一系列互相连接的节点组成的数据结构。通常会有一个节点称为*头节点*，其他节点顺序跟在头节点后面，最后一个节点称为*尾节点*。我们可以用指针轻松实现节点之间的连接，动态按需分配每个节点。

这种方法比节点的数组好，使用数组的结果就是创建固定数量的节点，而不管实际需要几个。节点之间的连接是用数组元素的索引实现的。使用数组不如使用动态内存分配和指针灵活。

比如说，如果我们想要改变数组元素的顺序，就需要复制两个元素，而结构体元素可能很大。此外，对于添加或删除元素，不论是为新元素腾出空间或是删除已有元素，都可能需要移动数组的一大部分。

链表有好几种类型，最简单的是单链表，一个节点到下一个节点只有一个连接，连接从头节点开始，到尾节点结束。循环链表没有尾节点，链表的最后一个节点又指向头节点。双链表用了两个链表，一个向前连接，一个向后连接，我们可以在两个方向上查找节点，这类链表更灵活，但是也更难实现。图 6-5 从理论上解释了这些类型的链表。

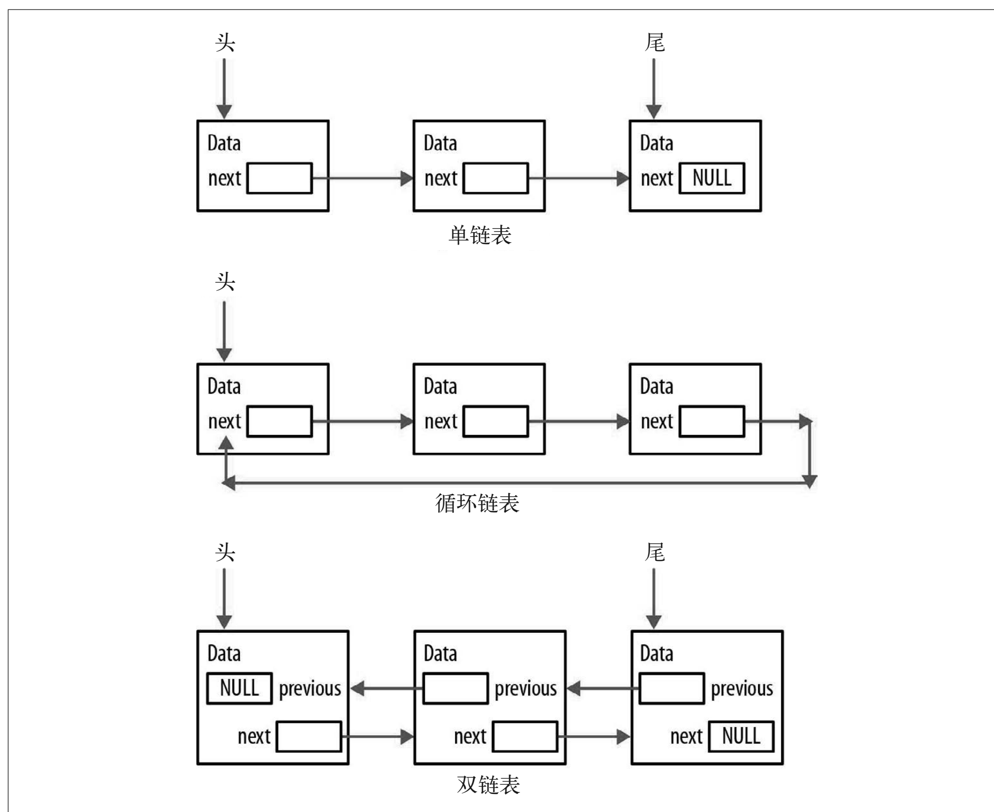


图 6-5：链表类型

这一节我们来看看如何创建和使用单链表，下面的代码显示了用来支持链表的结构体，Node 结构体定义一个节点，它有两个指针，第一个是 void 指针，持有任意类型的数据，第二个是指向下一个节点的指针。LinkedList 结构体表示链表，持有指向头节点和尾节点的指针，当前指针用来辅助遍历链表：

```
typedef struct _node {
    void *data;
    struct _node *next;
} Node;

typedef struct _linkedList {
    Node *head;
    Node *tail;
    Node *current;
} LinkedList;
```

我们会开发几个使用这些结构体支持链表功能的函数：

`void initializeList(LinkedList*)`	初始化链表
`void addHead(LinkedList*, void*)`	给链表的头节点添加数据
`void addTail(LinkedList*, void*)`	给链表的尾节点添加数据
`void delete(LinkedList*, Node*)`	从链表删除节点
`Node *getNode(LinkedList*, COMPARE, void*)`	返回包含指定数据的节点指针
`void displayLinkedList(LinkedList*, DISPLAY)`	打印链表

使用链表之前要先初始化，如下所示的 `initializeList` 函数执行这个任务，将 `LinkedList` 对象的指针传递给函数，函数把结构体里的指针置为 `NULL`：

```
void initializeList(LinkedList *list) {
    list->head = NULL;
    list->tail = NULL;
    list->current = NULL;
}
```

`addHead` 和 `addTail` 函数分别向链表的头和尾添加数据。在这个链表的实现中，`add` 和 `delete` 函数负责分配和释放链表节点用到的内存，这样就不需要链表用户负责了。

在如下所示的 `addHead` 函数中，先给节点分配内存，然后把传递给函数的数据赋给结构体的 `data` 字段。通过把 `data` 以 `void` 指针的形式传递，链表就能够持有用户想用的任何类型的数据了。

接下来，我们检查链表是否为空，如果为空，就把尾指针指向节点，然后把节点的 `next` 字段赋值为 `NULL`；如果不为空，那么将节点的 `next` 指针指向链表头。无论哪种情况，链表头都指向节点：

```
void addHead(LinkedList *list, void* data) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->data = data;
    if (list->head == NULL) {
        list->tail = node;
        node->next = NULL;
    } else {
        node->next = list->head;
    }
    list->head = node;
}
```

下面的代码片段展示了 `initializeList` 和 `addHead` 函数的用法。将三个雇员结构体添加到链表中，图 6-6 显示了执行这些语句后的内存分配情况。为了简化图，我们去掉了一些箭头，此外，还简化了 `Employee` 结构体的 `name` 数组。

```
LinkedList linkedList;

Employee *samuel = (Employee*) malloc(sizeof(Employee));
strcpy(samuel->name, "Samuel");
```

```
samuel->age = 32;

Employee *sally = (Employee*) malloc(sizeof(Employee));
strcpy(sally->name, "Sally");
sally->age = 28;

Employee *susan = (Employee*) malloc(sizeof(Employee));
strcpy(susan->name, "Susan");
susan->age = 45;

initializeList(&linkedList);

addHead(&linkedList, samuel);
addHead(&linkedList, sally);
addHead(&linkedList, susan);
```

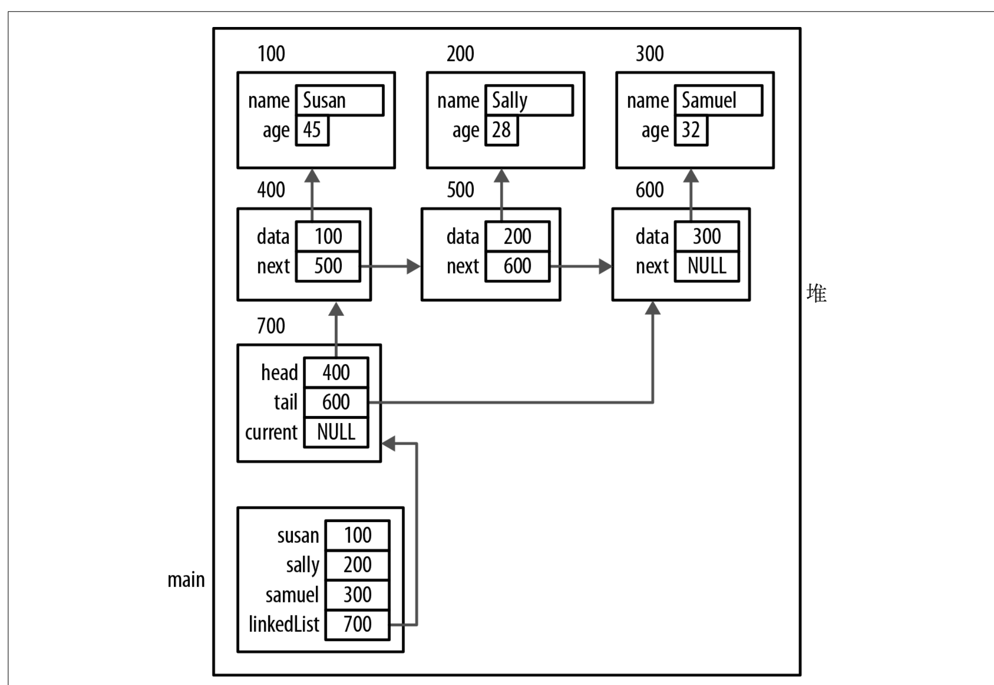


图 6-6：addHead 示例

接下来是 addTail 函数。它先为新节点分配内存，然后把数据赋给 data 字段。因为我们总是将节点添加到末尾，所以该节点的 next 字段被赋值为 NULL。如果链表为空，那么 head 指针就是 NULL，就可以把新节点赋给 head；如果不为空，那么就将尾节点的 next 指针赋为新节点。无论如何，我们会将链表的 tail 指针赋为该节点：

```
void addTail(LinkedList *list, void* data) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->data = data;
    node->next = NULL;
    if (list->head == NULL) {
        list->head = node;
    } else {
        list->tail->next = node;
    }
    list->tail = node;
}
```

下面的代码片段说明了 addTail 函数的用法。这里就不重复列出创建雇员对象的代码了，addTail 函数将雇员以跟上例相反的顺序添加进去，这样内存分配情况就跟图 6-6 一样。

```
initializeList(&linkedList);

addTail(&linkedList, susan);
addTail(&linkedList, sally);
addTail(&linkedList, samuel);
```

delete 函数从链表删除一个节点。为了简化这个函数，将删除节点的指针作为要传递的参数。函数的用户可能有数据的指针，但是没有持有数据的节点的指针，为了帮助定位节点，我们提供了一个辅助函数 getNode 来返回节点的指针。getNode 函数接受三个参数：

- 指向链表的指针；
- 指向比较函数的指针；
- 指向要查找的数据的指针。

下面是 getNode 函数的代码，变量 node 一开始指向链表头，然后我们遍历链表直到找到匹配的节点或者到达链表的末尾。我们调用 compare 函数来判断当前节点是否匹配。当两个数据相等时，它会返回 0。

```
Node *getNode(LinkedList *list, COMPARE compare , void* data) {
    Node *node = list->head;
    while (node != NULL) {
        if (compare(node->data, data) == 0) {
            return node;
        }
        node = node->next;
    }
    return NULL;
}
```

compare 函数说明了如何在运行时用函数指针来决定用哪个函数执行比较操作，这

样会给链表的实现增加可观的灵活性，因为我们不需要在 getNode 函数中硬编码比较函数的名字。

下面是 delete 函数，为了保持函数简单，它不会检查链表内的空值和传入的节点。第一个 if 语句处理删除头节点的情况。如果头节点是唯一的节点，那么将头节点和尾节点置为空值；否则，将头节点赋值为原头节点的下一个节点。

else 语句用 tmp 指针从头到尾遍历链表，不论是将 tmp 赋值为 NULL（表示要找的节点不在链表中），还是 tmp 的下一个节点就是我们要找的节点，while 循环都会结束。这是单链表，所以我们需要知道要删除的目标节点的前一节点是哪个，必需知道这一点才能把前一个节点的 next 字段赋值为目标节点的下一个节点。在 delete 函数的末尾，将节点释放。用户负责在调用 delete 函数之前删除节点指向的数据。

```
void delete(LinkedList *list, Node *node) {
    if (node == list->head) {
        if (list->head->next == NULL) {
            list->head = list->tail = NULL;
        } else {
            list->head = list->head->next;
        }
    } else {
        Node *tmp = list->head;
        while (tmp != NULL && tmp->next != node) {
            tmp = tmp->next;
        }
        if (tmp != NULL) {
            tmp->next = node->next;
        }
    }
    free(node);
}
```

下面的代码片段说明了这个函数的使用方法。将三个雇员添加到链表头上，我们用 6.4 节中提到的 compareEmployee 函数来进行比较操作：

```
addHead(&linkedList, samuel);
addHead(&linkedList, sally);
addHead(&linkedList, susan);

Node *node = getNode(&linkedList,
        (int (*)(void*, void*))compareEmployee, sally);
delete(&linkedList, node);
```

执行这段代码后程序栈和堆的状态如图 6-7 所示。

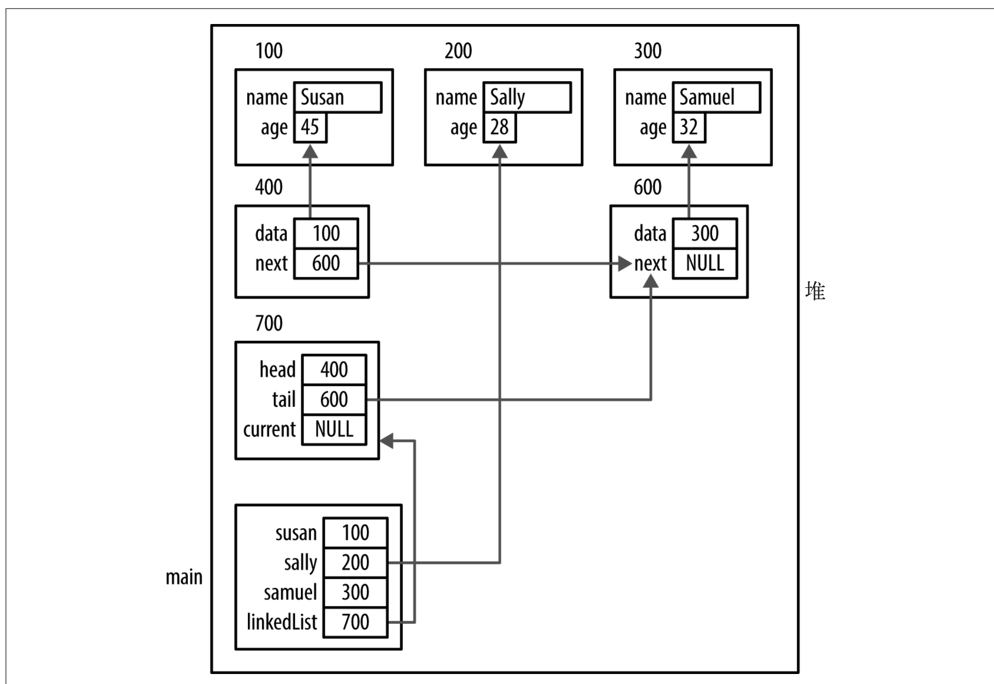


图 6-7：删除示例

如下所示的 displayLinkedList 函数说明如何遍历链表，从头节点开始，用第二个参数传入的函数打印每一个元素。将 next 字段的值赋给节点指针，打印最后一个节点后会结束：

```
void displayLinkedList(LinkedList *list, DISPLAY display) {
    printf("\nLinked List\n");
    Node *current = list->head;
    while (current != NULL) {
        display(current->data);
        current = current->next;
    }
}
```

下面说明这个函数用 6.4 节中开发的 displayEmployee 函数打印链表：

```
addHead(&linkedList, samuel);
addHead(&linkedList, sally);
addHead(&linkedList, susan);

displayLinkedList(&linkedList, (DISPLAY)displayEmployee);
```

这段代码的输出如下：

```
Linked List
Susan     45
Sally     28
Samuel    32
```

6.4.2 用指针支持队列

队列是一种线性数据结构，行为类似排队。它通常支持两种主要操作：入队和出队。入队操作把元素添加到队列中，出队操作从队列中删除元素。一般来说，第一个添加到队列中的元素也是第一个离开队列的元素，这种行为被称为先进先出（FIFO）。

实现队列经常用到链表。入队操作就是将节点添加到链表头，出队操作就是从链表尾删除节点。为了说明队列，我们会用到 6.4.1 节中开发的链表。

先用类型定义语句来定义队列，它基于链表，如下所示。现在可以用 `Queue` 来清晰地表达我们想要的东西了：

```
typedef LinkedList Queue;
```

要实现初始化操作，要做的只是利用 `initializeList` 函数。我们不会直接调用这个函数，而是使用下面的 `initializeQueue` 函数：

```
void initializeQueue(Queue *queue) {
    initializeList(queue);
}
```

类似地，下面的代码会用 `addHead` 函数向队列中添加一个节点：

```
void enqueue(Queue *queue, void *node) {
    addHead(queue, node);
}
```

之前实现的链表没有删除尾节点的函数，下面的 `dequeue` 函数删除最后一个节点，这里需要处理以下三种情况：

- 空队列

  返回 `NULL`

- 单节点队列

  由 `else if` 语句处理

- 多节点队列

  由 `else` 分支处理

在最后一种情况中，我们用 `tmp` 指针来一个节点一个节点地前进，直到它指向尾节点的前一个节点，然后按顺序执行下面三种操作：

(1) 将尾节点赋值为 `tmp`；
(2) 将 `tmp` 指针前进一个节点；
(3) 将尾节点的 `next` 字段置为 `NULL`，表示后面没有节点了。

必须按照这个顺序来确保链表的完整性，图 6-8 说明了原理，带圆圈的数字对应上面的三步操作。

```
void *dequeue(Queue *queue) {
    Node *tmp = queue->head;
    void *data;
    if (queue->head == NULL) {
        data = NULL;
    } else if (queue->head == queue->tail) {
        queue->head = queue->tail = NULL;
        data = tmp->data;
        free(tmp);
    } else {
        while (tmp->next != queue->tail) {
            tmp = tmp->next;
        }
        queue->tail = tmp;
        tmp = tmp->next;
        queue->tail->next = NULL;
        data = tmp->data;
        free(tmp);
    }
    return data;
}
```

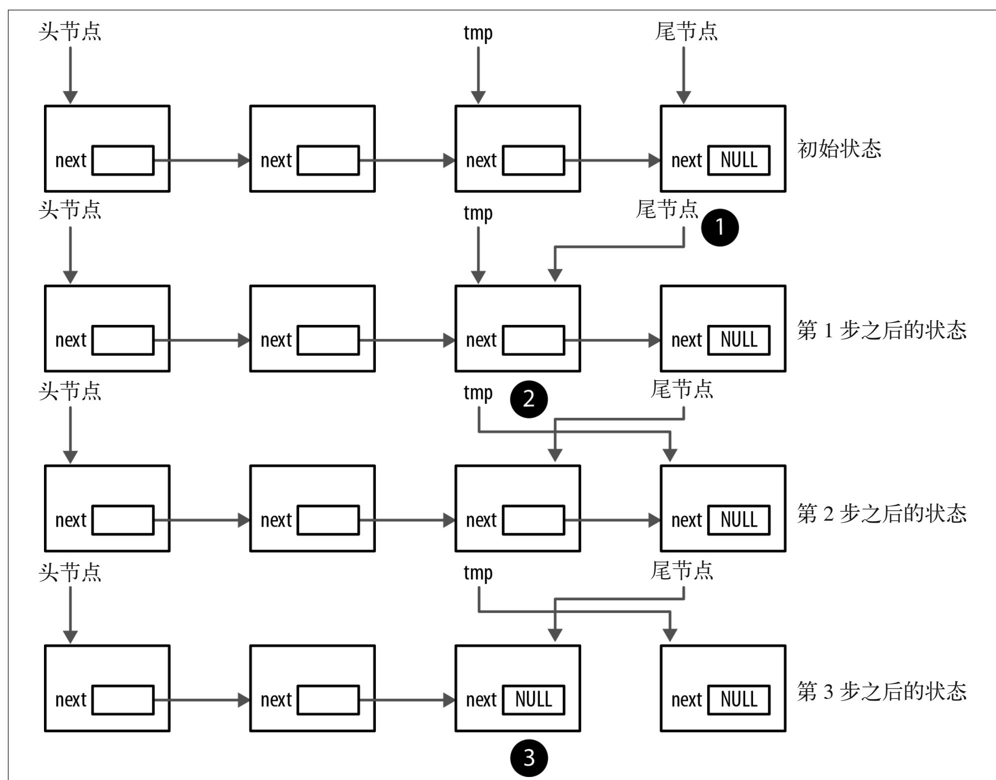


图 6-8：出队函数示例

返回赋给节点的数据后节点被释放。下面的代码片段用前面创建的雇员信息说明这些函数的用法：

```
Queue queue;
initializeQueue(&queue);

enqueue(&queue, samuel);
enqueue(&queue, sally);
enqueue(&queue, susan);

void *data = dequeue(&queue);
printf("Dequeued %s\n", ((Employee*) data)->name);
data = dequeue(&queue);
printf("Dequeued %s\n", ((Employee*) data)->name);
data = dequeue(&queue);
printf("Dequeued %s\n", ((Employee*) data)->name);
```

输出如下：

```
Dequeued Samuel
Dequeued Sally
Dequeued Susan
```

6.4.3 用指针支持栈

栈数据结构也是一种链表。对于栈，元素被推入栈顶，然后被弹出。当多个元素被推入和弹出时，栈的行为是先进后出（FILO）。第一个推入栈的元素最后一个弹出。

就像队列的实现，我们可以用链表来支持栈操作。最常见的两种操作是入栈和出栈。我们用 addHead 函数实现入栈操作，出栈操作需要增加一个新函数来删除头节点。我们先定义栈：

```
typedef LinkedList Stack;
```

要初始化栈，需要添加 initializeStack 函数，这个函数调用 initializeList 函数：

```
void initializeStack(Stack *stack) {
    initializeList(stack);
}
```

入栈操作调用 addHead 函数，如下所示：

```
void push(Stack *stack, void* data) {
    addHead(stack, data);
}
```

下面是出栈操作的实现，我们先把栈的头节点赋给一个节点指针，这里涉及三种情况。

- 栈为空

  函数返回 NULL。

- 栈中有一个元素

  如果节点指向尾节点，那么头节点和尾节点是同一个元素。将头节点和尾节点置为 NULL，然后返回数据。

- 栈中有多个元素

  在这种情况下，我们将头节点赋值为链表中的下一个元素，然后返回数据。

在后两种情况下，节点会被释放：

```
void *pop(Stack *stack) {
    Node *node = stack->head;
    if (node == NULL) {
        return NULL;
    } else if (node == stack->tail) {
        stack->head = stack->tail = NULL;
        void *data = node->data;
        free(node);
        return data;
    } else {
        stack->head = stack->head->next;
        void *data = node->data;
        free(node);
        return data;
    }
}
```

我们会重复利用 6.4.1 节中创建的雇员实例来说明栈的用法。下面的代码片段会把三个雇员入栈再出栈：

```
Stack stack;
initializeStack(&stack);

push(&stack, samuel);
push(&stack, sally);
push(&stack, susan);

Employee *employee;

for(int i=0; i<4; i++) {
    employee = (Employee*) pop(&stack);
    printf("Popped %s\n", employee->name);
}
```

执行后得到如下输出。因为我们调用了四次出栈函数，最后一次会返回 NULL：

```
Popped Susan
Popped Sally
Popped Samuel
Popped (null)
```

有时候还有其他的栈操作，如查看栈顶元素，它会返回栈顶的元素，但不会将其弹出。

6.4.4 用指针支持树

树是很有用的数据结构，它的名字源于元素之间的关系。通常，子节点连接到父节点，从整体上看就像一颗倒过来的树，根节点表示这种数据结构的开始元素。

树可以有任意数量的子节点，不过，二叉树比较常见，它的每个节点能有 0 个、1 个或是 2 个子节点。子节点要么是左子节点，要么是右子节点。没有子节点的节点称为叶子节点，就跟树叶一样。本节中的示例会阐明二叉树。

指针提供了一种维护三个节点之间关系的直观、动态的方式。我们可以动态分配节点，将其按需插入树中。这里使用下面的结构体作为节点，借助 `void` 指针可以处理我们需要的任意类型的数据：

```
typedef struct _tree {
    void *data;
    struct _tree *left;
    struct _tree *right;
} TreeNode;
```

按照特定顺序向树中插入节点是有意义的，这样可以让很多操作（比如搜索）变得容易。下面的顺序很常见：插入新节点后，这个节点的所有左子节点的值都比父节点小，所有右子节点的值都比父节点的值大，这样的树称为二叉查找树。

下面这个 `insertNode` 函数会把一个节点插入二叉查找树，不过，要插入节点，首先需要在新节点和已有节点之间做比较。我们用 `COMPARE` 函数指针来传递比较函数的地址，函数的第一部分为新节点分配内存并把数据赋给节点。因为新节点插入树后总是叶子节点，所以将左子节点和右子节点置为 `NULL`：

```
void insertNode(TreeNode **root, COMPARE compare, void* data) {
    TreeNode *node = (TreeNode*) malloc(sizeof(TreeNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    if (*root == NULL) {
        *root = node;
        return;
    }

    while (1) {
        if (compare((*root)->data, data) > 0) {
            if ((*root)->left != NULL) {
                *root = (*root)->left;
            } else {
                (*root)->left = node;
                break;
            }
        } else {
            if ((*root)->right != NULL) {
                *root = (*root)->right;
            } else {
```

```
                (*root)->right = node;
                break;
            }
        }
    }
}
```

首先，检查根节点来判断树是否为空。如果为空，那么就把新节点赋给根节点，函数返回。根节点是以 TreeNode 指针的指针的形式传递的，这是必要的，因为我们需要修改传入函数的指针，而不是指针指向的对象。我们已经在 1.4.1 节中详细讨论过多重间接引用了。

如果树非空，程序就进入一个无限循环，直到将新节点插入树中结束。每次循环迭代都会比较新节点和当前节点，根据比较结果，将局部 root 指针置为左子节点或者右子节点，这个 root 指针总是指向树的当前节点。如果左子节点或右子节点为空，那么就将新节点添加为当前节点的子节点，循环结束。

为了说明 insertNode 函数，我们会重用 6.4 节中创建的雇员实例。下面的代码片段初始化一个空的 TreeNode，然后插入三个雇员结构体，程序栈的结果和堆的状态如图 6-9 所示。为了简化这个图，我们省略了某些指向雇员结构体的线，并调整了节点在堆中的位置，以反映树结构的顺序：

```
TreeNode *tree = NULL;

insertNode(&tree, (COMPARE) compareEmployee, samuel);
insertNode(&tree, (COMPARE) compareEmployee, sally);
insertNode(&tree, (COMPARE) compareEmployee, susan);
```

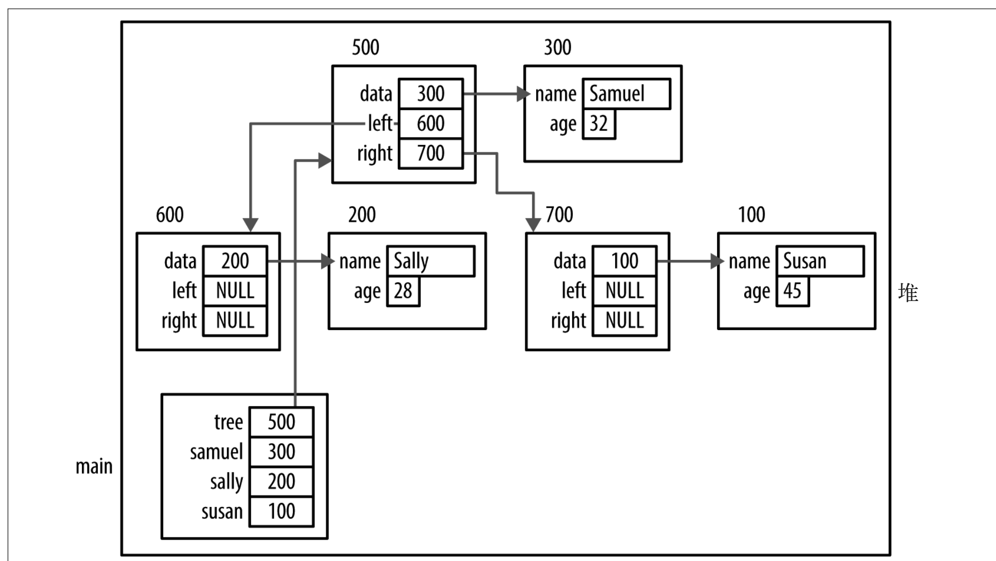


图 6-9：insertNode 函数

图 6-10 说明了这棵树的逻辑结构。

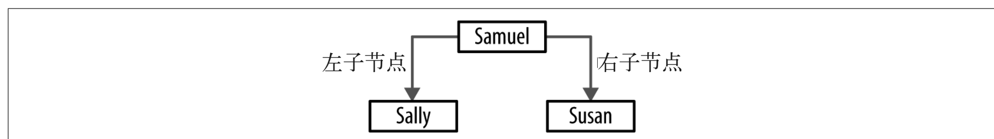


图 6-10：树的逻辑组织

二叉树可以满足多种需求，遍历二叉树的方式有三种：前序、中序和后序。三种技术步骤相同，但顺序不同。三个步骤如下：

- 访问节点
  处理节点
- 往左
  转移到左子节点
- 往右
  转移到右子节点

对我们来说，访问节点的目的是打印其内容。访问节点的三种顺序如下所示。

- 中序
  先往左，访问节点，再往右。
- 前序
  访问节点，往左，再往右。
- 后序
  先往左，再往右，最后访问节点。

函数的实现如下，所有的函数的参数都是树根和作为打印函数的一个函数指针，它们都是递归的。只要传入的根节点非空就会调用自身，不同点只在于执行三步操作的顺序：

```
void inOrder(TreeNode *root, DISPLAY display) {
    if (root != NULL) {
        inOrder(root->left, display);
        display(root->data);
        inOrder(root->right, display);
    }
}

void postOrder(TreeNode *root, DISPLAY display) {
```

```
        if (root != NULL) {
            postOrder(root->left, display);
            postOrder(root->right, display);
            display(root->data);
        }
    }

    void preOrder(TreeNode *root, DISPLAY display) {
        if (root != NULL) {
            display(root->data);
            preOrder(root->left, display);
            preOrder(root->right, display);
        }
    }
```

下面的代码片段调用这些函数：

```
preOrder(tree, (DISPLAY) displayEmployee);
inOrder(tree, (DISPLAY) displayEmployee);
postOrder(tree, (DISPLAY) displayEmployee);
```

表 6-1 显示的是基于前面初始化的树每次函数调用的输出。

表6-1：遍历技术

前序	Samuel 32 Sally 28 Susan 45
中序	Sally 28 Samuel 32 Susan 45
后序	Sally 28 Susan 45 Samuel 32

中序遍历会返回树成员的排序列表，前序和后序遍历跟栈和队列配合使用可以计算算术表达式。

6.5 小结

指针的强大和灵活在创建和支持数据结构的过程中体现得淋漓尽致。指针跟动态分配结构体内存相结合，确保了数据结构的创建对内存的使用高效、可伸缩，从而满足应用程序的需求。

本章一开始讨论了如何分配结构体的内存，需要注意的是，结构体字段之间和结构体数组之间可能存在填充。动态内存分配和释放的开销可能很大，我们研究了一种维护结构体池的技术来确保开销最小。

我们也讲到了几种常用的数据结构的实现，其中有几种使用链表支持。通过在运行时确定比较函数和打印函数，函数指针将这些实现变得更加灵活。

第7章
安全问题和指针误用

很少有应用程序不需要关注安全性和可靠性，而频繁的安全漏洞报告和应用程序故障会加强这种关注。巩固应用程序安全性的责任基本落在了开发者身上。在本章中，我们会研究让应用程序更安全可靠的实践。

因为C的某些特性，用C写安全的应用程序跟用其他语言有所不同。比如说，C不会阻止程序员越过数组边界写入，这样会导致内存损坏，也会引发安全风险。此外，误用指针通常也是很多安全问题的根本原因。

如果应用程序的行为和预期不一致，这可能不是安全问题，至少未授权访问就是这种情况。不过，有时候这种行为可能会被利用，从而导致拒绝服务，进而危害应用程序。误用指针导致的非预期行为已经在本书其他部分讲过了，本章还会讲到更多的指针误用。

CERT组织（http://www.cert.org/）是了解C和其他语言安全问题全面解决方案的好来源。这个组织研究互联网安全漏洞，我们主要关注跟指针相关的安全问题。CERT组织研究的很多安全问题都能追根溯源到指针的误用。理解指针和使用指针的恰当方法是开发安全可靠的应用程序的重要工具。其中部分主题已经在前面的章节中提到过了，可能不是从安全角度而是从编程实践的角度出发的。

操作系统（OS）已经引入了一些安全改进，有些改进反映在内存的使用方式上。尽管这些改进通常超出了开发者的控制范围，但是它们确实会影响程序。理解这些问题有助于解释应用程序的行为。我们会把精力集中在地址空间布局随机化和数据执行保护上。

地址空间布局随机化（Address Space Layout Randomization，ASLR）过程会把应用程序的数据区域随机放置在内存中，这些数据区域包括代码、栈和堆。随机放置这些区域导致攻击者更难预测内存的位置，从而更难利用它们。有些类型的攻击（比如说 return-to-libc 攻击），会覆写栈的一部分，然后把控制转移到这个区域。这个区域经常是共享 C 库 libc。如果栈和 libc 的位置是未知的，这类攻击的成功率就会降低。

如果代码位于内存的不可执行区域，数据执行保护（Data Execution Prevention，DEP）技术会阻止执行这些代码。在有些类型的攻击中，恶意代码会覆写内存的某个区域，然后将控制转移到这个区域。如果这个区域（比如栈或是堆）的代码不可执行，那么恶意代码就无法执行了。这种技术可以用硬件实现，也可以用软件实现。

本章会从以下几个角度研究安全问题：

- 声明和初始化指针；
- 误用指针；
- 释放问题。

7.1 指针的声明和初始化

在声明和初始化指针时可能会出现问题，更准确地说，指针初始化失败。在本节中，我们会研究出现这类问题的情况。

7.1.1 不恰当的指针声明

考虑如下声明：

```
int* ptr1, ptr2;
```

声明本身没错，不过，可能跟我们的本意不同，它把 ptr1 声明为整数指针，而把 ptr2 声明为整数。我们将星号有意紧挨着数据类型，而 ptr1 前面则加了空格。这样的位置对编译器来说没有区别，但是对于读代码的人来说，可能暗示着 ptr1 和 ptr2 都是整数指针。然而，只有 ptr1 是指针。

在同一行中把两个变量都声明为指针的正确写法如下所示：

```
int *ptr1, *ptr2;
```



每个变量声明独占一行更好。

用类型定义代替宏定义是另一个好习惯。类型定义允许编译器检查作用域规则，而宏定义不一定会。

我们可以使用宏指令辅助声明变量，如下所示。在这里，define 指令包装了整数指针，并用它来声明变量：

```
#define PINT int*
PINT ptr1, ptr2;
```

不过，结果就跟前面所说的一样。更好的方法是用下面的类型定义：

```
typedef int* PINT;
PINT ptr1, ptr2;
```

两个变量均被声明为整数指针。

7.1.2 使用指针前未初始化

在初始化指针之前就使用指针会导致运行时错误，有时候将这种指针称为野指针。下面这个简单的例子声明了一个整数指针，但是使用之前没有为其赋值：

```
int *pi;
...
printf("%d\n",*pi);
```

图 7-1 说明了此时的内存分配情况。

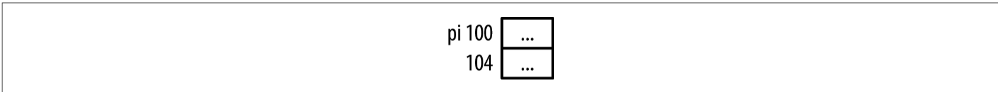


图 7-1：野指针

这里没有初始化 pi 变量，因此它会包含垃圾数据（在图中用省略号表示）。如果 pi 中的内存地址超出了应用程序的合法地址空间，这段代码很可能会在执行过程中终止。否则打印出来的就是恰好位于那个地址的数据（不管具体是什么），而且会表示为整数。如果我们用的是字符串指针，就经常会看到打印出奇怪的字符（直到遇到末尾的 0 才停止）。

7.1.3 处理未初始化指针

指针本身并不能告诉我们它是否有效，所以，不能只靠检查指针的内容来判断它是否有效。不过，有三种方法可以用来处理未初始化的指针：

- 总是用 NULL 来初始化指针；
- 用 assert 函数；
- 用第三方工具。

把指针初始化为 NULL 更容易检查是否使用正确。即便这样，检查空值也比较麻烦，如下所示：

```
int *pi = NULL;
...
if(pi == NULL) {
   // 不应该解引 pi
} else {
   // 可以使用 pi
}
```

我们可以用 assert 函数来测试指针是否为空值。下例测试了 pi 变量是否为空值。如果表达式为真，那么什么都不会发生，如果表达式为假，程序会终止。这样，指针为空的话程序就会终止。

```
assert(pi != NULL);
```

对于应用程序的调试版本，这种方法可能可以接受。如果指针是空值，输出类似下面：

```
Assertion failed: pi != NULL
```

我们在 assert.h 头文件中声明 assert 函数。

可使用第三方工具来帮助定位这类问题，此外，有些编译器选项也比较有用，7.4 节中会讲到。

7.2 指针的使用问题

在本节中，我们会研究解引操作和数组下标的误用，也会研究跟字符串、结构体和函数指针有关的问题。

很多安全问题聚焦的是缓冲区溢出的概念。覆写对象边界以外的内存就会导致缓冲区溢出，这块内存可能是本程序的地址空间，也可能是其他进程的，如果是程序地址空间以外的内存，大部分操作系统会发出一个段错误然后终止程序。因为这个原因恶意引发的终止本身就是拒绝服务攻击，这类攻击不会获取未授权的访问，但会试图搞垮应用程序甚至是服务器。

如果缓冲区溢出发生在应用程序的地址空间内，就会导致对数据的未授权访问和

（或）控制转移到其他代码段，于是就可能攻陷系统。以超级用户权限运行应用程序更要加倍小心。

下面几种情况可能导致缓冲区溢出：

- 访问数组元素时没有检查索引值；
- 对数组指针做指针算术运算时不够小心；
- 用 gets 这样的函数从标准输入读取字符串；
- 误用 strcpy 和 strcat 这样的函数。

如果缓冲区溢出发生在栈帧的元素上，就可能把栈帧的返回地址部分覆写为对同一时间创建的恶意代码的调用。查看 3.1 节来获取关于栈帧的详细信息。函数返回时会将控制转移到恶意函数，该函数可以执行任何操作，只受限于当前用户的特权等级。

7.2.1　测试NULL

用 malloc 这类函数时一定要检查返回值，否则可能会导致程序非正常终止。下面说明一般的方法：

```
float *vector = malloc(20 * sizeof(float));
if(vector == NULL) {
    // malloc 分配内存失败
} else {
    // 处理 vector
}
```

7.2.2　错误使用解引操作

声明和初始化指针的常用方法如下：

```
int num;
int *pi = &num;
```

下面是一种看似等价的声明方法：

```
int num;
int *pi;
*pi = &num;
```

不过，这样是错误的，注意最后一行的解引操作。我们试图把 num 的地址赋给 pi 所指向的内存地址（而不是 pi）。指针 pi 还没有被初始化。我们犯了一个简单的错误，误用了解引操作，正确的写法如下：

```
int num;
int *pi;
pi = &num;
```

在原声明 `int *pi = &num` 中，星号把变量声明为指针，而不是解引操作。

7.2.3 迷途指针

释放指针后却仍然在引用原来的内存，就会产生迷途指针，这个问题已经在 2.4 节中详细讲过了。如果之后试图访问这块内存，其内容可能已经改变。对这块内存进行写操作可能会损坏内存，而读操作则可能返回无效数据，这两种情况都可能导致程序终止。

直到最近，这才被认为是一个安全问题。正如迷途指针（https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf）中所讲的那样，存在利用迷途指针的可能性。不过，这种方法建立在利用 C++ 中的 VTable（虚表）的前提下。虚表是函数指针的数组，用来支持 C++ 的虚方法。除非使用涉及函数指针的类似方法，否则在 C 中这应该不会有问题。

7.2.4 越过数组边界访问内存

没有什么可以阻止程序访问为数组分配的空间以外的内存。在本例中，我们声明并初始化了三个数组来说明这种行为。假设数组分配在连续的内存位置。

```
char firstName[8] = "1234567";
char middleName[8] = "1234567";
char lastName[8] = "1234567";

middleName[-2] = 'X';
middleName[0] = 'X';
middleName[10] = 'X';

printf("%p %s\n",firstName,firstName);
printf("%p %s\n",middleName,middleName);
printf("%p %s\n",lastName,lastName);
```

为了说明如何覆写内存，将三个数组初始化为简单的数字。程序的行为会随着编译器和机器而变化，但是这段代码应该能正常运行并覆写 `firstName` 和 `lastName` 中的字符，输出如下。图 7-2 说明了内存分配情况。

```
116 12X4567
108 X234567
100 123456X
```

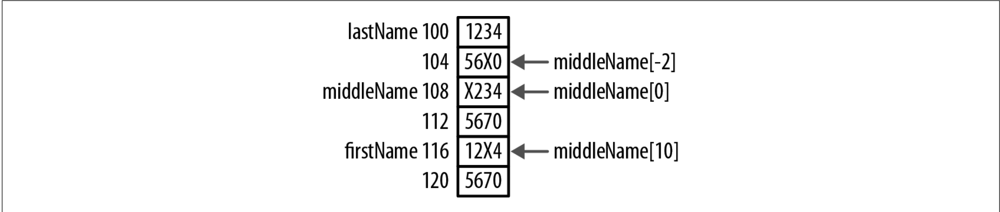


图 7-2：使用无效的数组索引

第 4 章解释过，用下标计算的地址不会检查索引值，这就是一个简单的缓冲区溢出。

7.2.5　错误计算数组长度

将数组传递给函数时，一定要同时传递数组长度。这个信息帮助函数避免越过数组边界。在下面的 `replace` 函数中，将字符串的地址随着替换用的字符以及缓冲区长度一块传入，函数的目的是把字符串中所有的字符都替换为传入的字符，直到 `NUL` 字符。长度参数防止函数越过缓冲区写入：

```
void replace(char buffer[], char replacement, size_t size) {
    size_t count = 0;
    while(*buffer != NUL && count++<size) {
        *buffer = replacement;
        buffer++;
    }
}
```

在下面的代码中，`name` 数组最多只能装 7 个字符再加上 `NUL` 字符。不过，我们有意越过数组边界写入来说明 `replace` 函数的工作原理。我们给 `replace` 函数传递了 `name` 和替换字符 `+`：

```
char name[8];
strcpy(name,"Alexander");
replace(name,'+',sizeof(name));
printf("%s\n", name);
```

执行代码后得到如下输出：

```
++++++++r
```

只是向数组添加了 8 个加号，`strcpy` 函数允许缓冲区溢出，但是 `replace` 函数不允许，前提还是假设传入的长度信息有效。要谨慎使用 `strcpy` 这类不传递缓冲区长度的函数。传递缓冲区长度能提供额外的安全屏障。

7.2.6　错误使用sizeof操作符

错误使用 sizeof 操作符的一个例子是试图检查指针边界但方法错误。下例为整数数组分配内存，然后把每个元素初始化为 0：

```
int buffer[20];
int *pbuffer = buffer;
for(int i=0; i<sizeof(buffer); i++) {
    *(pbuffer++) = 0;
}
```

不过，sizeof(buffer) 表达式返回了 80，因为缓冲区长度以字节计是 80（20 乘以 4 字节每元素）。for 循环执行了 80 次而不是 20 次，这很可能会导致内存访问异常，从而终止应用程序。可以在 for 表达式的测试条件中用 sizeof(buffer)/sizeof(int) 来避免这个问题。

7.2.7　一定要匹配指针类型

总是用合适的指针类型来装数据是个好主意。为了说明可能存在的陷阱，考虑下面的代码。将一个整数指针赋值给一个短整数指针：

```
int num = 2147483647;
int *pi = &num;
short *ps = (short*)pi;
printf("pi: %p Value(16): %x Value(10): %d\n", pi, *pi, *pi);
printf("ps: %p Value(16): %hx Value(10): %hd\n",
        ps, (unsigned short)*ps, (unsigned short)*ps);
```

这段代码的输出如下：

```
pi: 100 Value(16): 7fffffff Value(10): 2147483647
ps: 100 Value(16): ffff Value(10): -1
```

注意，看起来地址 100 处的第一个十六进制数字要么是 7，要么是 f，这取决于它是以整数还是短整数显示。这个明显的矛盾是在小字节序机器上运行代码的结果。图 7-3 说明了地址 100 处的常量的内存布局。

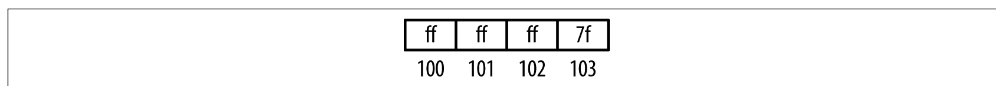


图 7-3：不匹配的指针类型

如果我们把它当做短整数，那就只用前两个字节，于是就得到了短整数值 -1。如果我们把它当做整数，就会用 4 个字节，于是得到 2 147 483 647。这类微妙的问题正是导致 C 和指针如此难的原因。

7.2.8 有界指针

有界指针是指指针的使用被限制在有效的区域内。比如说，现在有一个 32 个元素的数组，禁止对这个数组使用的指针访问数组前面或后面的任何内存。

C 没有对这类指针提供直接支持。不过，程序员可以显式地确保这个限制得以执行，如下所示：

```
#define SIZE 32

char name[SIZE];
char *p = name;
if(name != NULL) {
    if(p >= name && p < name+SIZE) {
        // 有效指针，继续
    } else {
        // 无效指针，错误分支
    }
}
```

这种方法比较麻烦，相较而言，7.4 节中讨论的静态分析可能会比较有用。

一种有趣的变化是创建一个指针检验函数（https://www.securecoding.cert.org/confluence/display/seccode/MEM10-C.+Define+and+use+a+pointer+validation+function），要这么做，必须知道初始的位置和范围。

另一种方法是利用 ANSI-C 和 C++ 的边界模型检查工具（CBMC，http://www.cprover.org/cbmc/）。这个应用程序会对 C 程序做一些安全问题检查，然后发现数组边界和缓冲区溢出的问题。



C++ 中的智能指针提供了一种模仿指针同时支持边界检查的方法，不幸的是，C 没有智能指针。

7.2.9 字符串的安全问题

字符串相关的安全问题一般发生在越过字符串末尾写入的情况。在本节中，我们主要关注可能造成这种问题的“标准”函数。

如果使用 `strcpy` 和 `strcat` 这类字符串函数，稍不留神就会引发缓冲区溢出。已经有人提出一些方法来取代，但都没有得到广泛认可。`strncpy` 和 `strncat` 函数可以对这种操作提供一些支持，它们的 `size_t` 参数指定要复制的字符的最大数量。不过，如果字符数量计算不正确，替代函数也容易出错。

C11 中（Annex K）加入了 strcat_s 和 strcpy_s 函数，如果发生缓冲区溢出，它们会返回错误，目前只有 Microsoft Visual C++ 支持。下面这个例子说明了 strcpy_s 函数的使用，它接受三个参数：目标缓冲区、目标缓冲区的长度以及源缓冲区。如果返回值是 0 就表示没有错误发生。不过在本例中会有错误发生，因为源缓冲区太大了，目标缓冲区装不下：

```
char firstName [8];
int result;
result = strcpy_s(firstName,sizeof(firstName),"Alexander");
```

还有 scanf_s 和 wscanf_s 函数可以用来防止缓冲区溢出。

gets 函数从标准输入读取一个字符串，并把字符保存在目标缓冲区中，它可能会越过缓冲区的声明长度写入。如果字符串太长的话，就会发生缓冲区溢出。

有些 Linux 系统也支持 strlcpy 和 strlcat 函数，但 GNU C 库不支持。有人认为这两个函数制造的问题比解决的还多，而且文档不全。

使用某些函数可能造成攻击者用*格式化字符串攻击*的方法访问内存。在这类攻击中，将用户提供的格式化字符串（如下所示）打造得可以访问内存，甚至能够注入代码。在这个简单的程序中，我们将第二个命令行参数作为 printf 函数的第一个参数：

```
int main(int argc, char** argv) {
    printf(argv[1]);
    ...
}
```

这个程序可以用类似于下面的命令执行：

```
main.exe "User Supplied Input"
```

输出类似于：

```
User Supplied Input
```

程序本身无害，但是精巧的攻击真的可以造成损害。这里不会就这个话题展开，不过，如何实现这样的攻击可以在 hackerproof.org 上找到。

printf、fprintf、snprintf 和 syslog 这些函数都接受格式化字符串作为参数，避免这类攻击的一种简单方法是永远不要把用户提供的格式化字符串传给这些函数。

7.2.10 指针算术运算和结构体

我们应该只对数组使用指针算术运算，因为数组肯定分配在连续的内存块上，指针

算术运算可以得到有效的偏移量。不过，不应该将它们用在结构体内，因为结构体的字段可能分配在不连续的内存区域。

下面这个结构体说明了这一点。为 name 字段分配 10 字节，之后是一个整数。然而，整数是对齐到 4 字节边界的，所以两个字段之间会有空隙。这类空隙在 6.1 节的“结构体的内存如何分配”中解释过了。

```
typedef struct _employee {
    char name[10];
    int age;
} Employee;
```

下面的代码试图用指针来访问结构体的 age 字段：

```
Employee employee;
// 初始化 employee
char *ptr = employee.name;
ptr += sizeof(employee.name);
```

指针包含地址 110，这是两个字段之间的 2 字节的地址，解引指针会把地址 110 处的 4 字节当做整数，如图 7-4 所示。

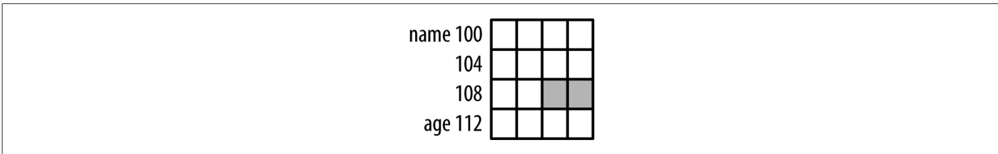
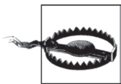


图 7-4：结构体填充示例



误用对齐的指针可能会导致程序非正常终止或是取到错误数据。此外，如果编译器需要生成额外的机器码来弥补不恰当的对齐，那么指针访问也可能变慢。

即使结构体内的内存是连续的，用指针算术运算来访问结构体的字段也不是好做法。下面的结构体定义了由三个整数组成的 Item，通常会将三个整数字段分配在连续的内存位置，不过也不一定：

```
typedef struct _item {
    int partNumber;
    int quantity;
    int binNumber;
}Item;
```

下面的代码片段声明了一个部件，然后用指针算术运算访问每个字段：

```
Item part = {12345, 35, 107};
int *pi = &part.partNumber;
printf("Part number: %d\n",*pi);
pi++;
printf("Quantity: %d\n",*pi);
pi++;
printf("Bin number: %d\n",*pi);
```

通常，输出就是我们所期望的那样，但也有例外。更好的办法是把每个字段赋给 pi：

```
int *pi = &part.partNumber;
printf("Part number: %d\n",*pi);
pi = &part.quantity;
printf("Quantity: %d\n",*pi);
pi = &part.binNumber;
printf("Bin number: %d\n",*pi);
```

更好的办法是根本不用指针，如下所示：

```
printf("Part number: %d\n",part.partNumber);
printf("Quantity: %d\n",part.quantity);
printf("Bin number: %d\n",part.binNumber);
```

7.2.11 函数指针的问题

函数和函数指针用来控制程序的执行顺序，但是它们可能会被误用，导致不可预期的行为。考虑下面的 getSystemStatus 函数的使用，它返回反应系统状态的整数：

```
int getSystemStatus() {
    int status;
    ...
    return status;
}
```

下面是判断系统状态是否为 0 的最好方法：

```
if(getSystemStatus() == 0) {
    printf("Status is 0\n");
} else {
     printf("Status is not 0\n");
}
```

接下来这个例子中，忘记了加上括号，这段代码不会正常执行：

```
if(getSystemStatus == 0) {
    printf("Status is 0\n");
} else {
    printf("Status is not 0\n");
}
```

系统会一直执行 else 分支，在逻辑表达式中，我们把函数的地址和 0 作比较，而

不是调用函数后比较返回值和 0。记住，只用函数名本身时返回的是函数的地址。

一个类似的错误是直接使用函数返回值，而不会将它的结果和其他值进行比较，这样会返回地址然后计算真假，而函数的地址不大可能是 0，结果就是返回的地址计算为真，因为 C 把非 0 值都作为真：

```
if(getSystemStatus) {
    // 永远为真
}
```

我们应该像下面这样写函数调用来判断状态是否为 0：

```
if(getSystemStatus()) {
```

如果函数和函数指针的签名不同，不要把函数赋给函数指针，这样会导致未定义的行为，一个误用的例子如下所示：

```
int (*fptrCompute)(int,int);
int add(int n1, int n2, int n3) {
    return n1+n2+n3;
}

fptrCompute = add;
fptrCompute(2,5);
```

我们试图只用两个参数调用 `add` 函数，而该函数期望的是三个参数，代码能编译，但是输出是不确定的。

函数指针可以执行不同的函数，这取决于分配给它的地址。比如说，我们可能想为一般的操作使用 `printf` 函数，但是有时候为了打印特定日志需要换成别的函数，那么可以像下面这样声明并使用函数指针：

```
int (*fptrIndirect)(const char *, ...) = printf;
fptrIndirect("Executing printf indirectly");
```

攻击者可能用缓冲区溢出来覆写函数指针的地址，如果发生这类攻击，控制可能会转移到内存中的任意位置。

7.3 内存释放问题

即便已经释放了内存，我们也不一定要用完指针或已释放的内存。有一个问题是：如果将同一块内存释放两次会发生什么。此外，一旦释放内存，我们可能就得保护留下的数据了。本节就来研究这几个问题。

7.3.1 重复释放

将同一块内存释放两次称为重复释放，2.3.2 节中已经解释过了。下面说明这种问题如何发生：

```
char *name = (char*)malloc(...);
...
free(name);     // 第一次释放
...
free(name);     // 第二次释放
```

在 zlib 压缩库的早期版本中就可能存在重复释放导致的拒绝服务攻击或是代码注入。不过这种情况很少发生，而且新版本已经修复了漏洞。关于这个漏洞的更多信息可以查看 cert.org。

避免这类漏洞的简单办法是释放指针后总是将其置为 NULL，大部分堆管理器都会忽略后续对空指针的释放。

```
char *name = (char*)malloc(...);
...
free(name);
name = NULL;
```

在 3.2.7 节中，我们开发了一个函数来实现这种效果。

7.3.2 清除敏感数据

一旦不再需要内存中的敏感数据，马上进行覆写是个好主意。当应用程序终止后，大部分操作系统都不会把用到的内存清零或是执行别的操作。系统可能会将之前用过的空间分配给别的程序，那么它就能访问内存中的内容。覆写敏感数据后别的应用程序就难以从之前持有这部分数据的内存中获取有用的信息。下面的代码会把程序中的敏感数据清空：

```
char name[32];
int userID;
char *securityQuestion;

// 赋值
...

// 删除敏感信息
memset(name,0,sizeof(name));
userID = 0;
memset(securityQuestion,0,strlen(securityQuestion));
```

如果声明 name 为指针，那么我们就应该在释放内存之前将其清空，如下所示：

```
char *name = (char*)malloc(...);
...
memset(name,0,sizeof(name));
free(name);
```

7.4 使用静态分析工具

有很多静态分析工具可以检查指针的误用，此外，大部分编译器都有选项来监测本章提到的很多问题。比如说，GCC 编译器的 `-Wall` 选项可以启用编译器警告。

下面说明本章的一些示例会产生什么样的警告，这里我们忘记在调用函数时写上括号了：

```
if(getSystemStatus == 0) {
```

结果会产生如下警告：

```
warning: the address of 'getSystemStatus' will never be NULL
```

下面是一个本质上完全一样的错误：

```
if(getSystemStatus) {
```

不过，警告信息会有所不同：

```
warning: the address of 'getSystemStatus' will always evaluate as 'true'
```

使用不兼容的指针类型也会产生警告：

```
int (*fptrCompute)(int,int);
int addNumbers(int n1, int n2, int n3) {
    return n1+n2+n3;
}

    ...
    fptrCompute = addNumbers;
```

下面是警告信息：

```
warning: assignment from incompatible pointer type
```

没有初始化指针通常也会出问题：

```
char *securityQuestion;
strcpy(securityQuestion,"Name of your home town");
```

产生的警告信息相当直白：

```
warning: 'securityQuestion' is used uninitialized in this function
```

还有很多静态分析工具可用，有的免费，有的收费，它们一般都会提供比编译器更强的诊断功能。因为太过复杂，超出了本书的范围，这里就不再举例了。

7.5 小结

本章研究了指针如何影响应用程序的安全性和可靠性。这些问题都是围绕声明和初始化指针、使用指针和释放内存组织的。比如说，在使用指针之前初始化很重要，用完字符串之后清空内存也很重要。将指针置为 NULL 是很多情况下的有效解决方法。

有不少误用指针的方法，其中多种涉及越过字符串边界写入内存，它是缓冲区溢出的一种形式。误用指针会导致很多方面的未定义行为，包括指针类型不匹配和不正确的指针算术运算。

我们说明了避免这类问题的一些技术，其中一部分只需要理解如何使用指针和字符串就能搞定。我们也接触了如何用编译器和静态分析工具来定位潜在的问题。

第 8 章

其他重要内容

几乎在任何方面，指针对 C 都至关重要，其中很多我们已经讲得很清楚了，比如数组和函数。本章研究一些不大适合放进前面章节但很重要的主题，这些主题将完善你对指针工作原理的理解。

本章将研究以下几个跟指针相关的主题：

- 指针的类型转换；
- 访问硬件设备；
- 别名和强别名；
- 使用 `restrict` 关键字；
- 线程；
- 面向对象技术。

关于线程，我们对两个方面感兴趣：一是用指针在线程之间共享数据这个基本问题，二是如何用指针支持回调函数。一个操作可能会调用某函数来执行任务，如果实际被调用的函数发生了改变，我们就称之为回调函数。比如第 5 章用到的 `sort` 函数就是一个回调函数。我们可以使用回调函数在线程之间通信。

本章会讲到两种在 C 中支持面向对象类型的方法：第一种是用不透明指针，这种技术对用户隐藏了数据结构的实现细节；第二种技术说明如何在 C 中实现多态类型。

8.1　转换指针

类型转换是一种基本操作，跟指针结合使用时很有用。转换指针对我们大有帮助，原因包括：

- 访问有特殊目的的地址；
- 分配一个地址来表示端口；
- 判断机器的字节序。

我们也会处理一个跟 8.2.1 节中的类型转换紧密相关的主题。



机器的字节序一般是指数据类型内部的字节顺序。两种常见的字节序是小字节序和大字节序。小字节序是指将低位字节存储在低地址中，而大字节序是指将高位字节存储在低地址中。

我们可以把整数转换成整数指针，如下所示：

```
int num = 8;
int *pi = (int*)num;
```

不过，一般来说这不是好实践，因为它允许你访问任意地址，包括系统不允许程序访问的位置。图 8-1 说明了这一点，地址 8 不在应用程序的地址空间内，如果解引指针，一般就会导致应用程序终止。

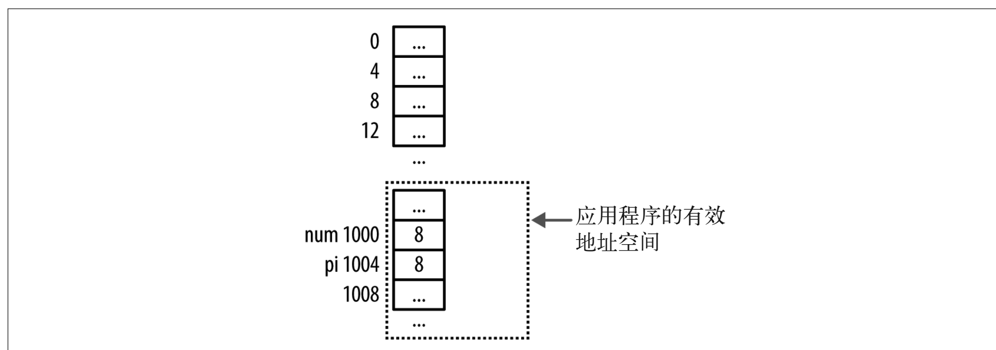


图 8-1：把整数转换成非法地址

有些情况下，比如我们需要寻址内存地址 0，就可能需要把指针转换成整数，然后再转换回指针，这在老式的系统上比较常见，其指针长度和整数长度相同。不过，有时候这样不能正常工作。下面说明了这种方法，输出跟实现相关：

```
pi = &num;
printf("Before: %p\n",pi);
int tmp = (int)pi;
pi = (int*)tmp;
printf("After: %p\n",pi);
```

把指针转换为整数再转换回指针从来就不是什么好办法，如果确实需要这么做，考虑用联合体，8.2.1 节会讨论这个主题。

记住在指针与整数之间来回转换和在指针与 `void` 指针之间来回转换不同，1.1.8 节中的“void 指针”部分有说明。



有时候容易将句柄和指针搞混。句柄是系统资源的引用，对资源的访问通过句柄实现。不过，句柄一般不提供对资源的直接访问，指针则包含了资源的地址。

8.1.1 访问特殊用途的地址

访问特殊用途的地址的需求一般发生在嵌入式系统上，嵌入式系统对应用程序的介入很少。比如说，在有些底层操作系统内核中，PC 的显存地址是 0xB8000，这个地址装的是字符模式下显示的第一行第一列的字符，我们可以把这个地址赋给某个指针，然后把某个字符赋给这个地址，代码如下所示。图 8-2 显示了内存布局。

```
#define VIDEO_BASE 0xB8000
int *video = (int *) VIDEO_BASE;
*video = 'A';
```

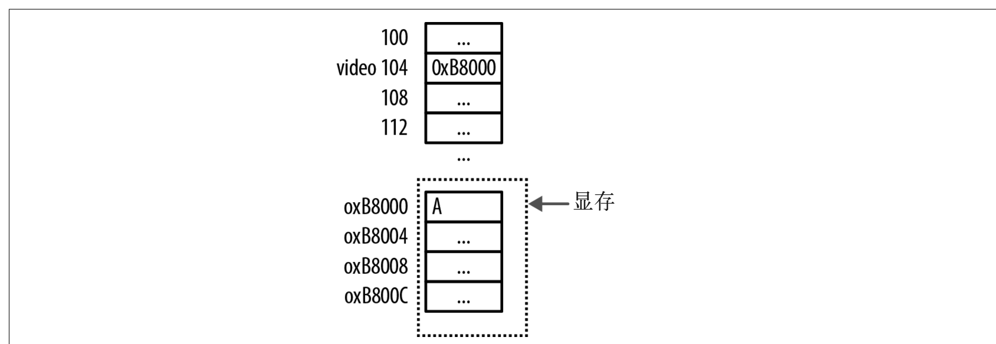


图 8-2：在 PC 上寻址显存

在适当情况下，可以读取这个地址的内容，不过一般不会对显存地址这么做。

当你需要寻址地址 0 的内存时，有时候编译器会把它当做 NULL 指针值。底层内核程序通常需要访问地址 0，有几种技术可以处理这种情况：

- 把指针置为 0（不一定能工作）；
- 把整数置为 0，再把这个整数转换为指针；
- 用 8.2.1 节中提到的联合体；
- 用 memset 函数把指针置为 0。

下面是一个使用 memset 函数的例子，这里将 ptr 引用的内存置为 0：

```
memset((void*)&ptr, 0, sizeof(ptr));
```

在需要寻址 0 地址内存的系统上，厂商一般会有解决问题的办法。

8.1.2 访问端口

端口既是硬件概念，也是软件概念。服务器用软件端口指明它们要接收发给这台机器的某类消息。硬件端口通常是一个连接到外部设备的物理输入输出系统组件。程序通过读写硬件端口可以处理信息和命令。

访问端口的软件一般是操作系统的一部分，下面说明如何用指针访问端口：

```
#define PORT 0xB0000000
unsigned int volatile * const port = (unsigned int *) PORT;
```

机器用十六进制地址表示端口，将数据作为无符号整数处理。volatile 关键字修饰符表示可以在程序以外改变变量。比如说，外部设备可能会向端口写入数据，且可以独立于计算机的处理器执行这个写操作。出于优化目的，编译器有时候会临时使用缓存或是寄存器来持有内存中的值，如果外部的操作修改了这个内存位置，改动并不能反映到缓存或寄存器中。

用 volatile 关键字可以阻止运行时系统使用寄存器暂存端口值，每次访问端口都需要系统读写端口，而不是从寄存器中读取一个可能已经过期的值。我们不应该把所有变量都声明为 volatile，因为这样会阻碍编译器进行所有类型的优化。

之后应用程序可以通过解引端口指针来读写端口，如下所示。内存布局见图 8-3，可以通过 0xB0000000 处的内存读写外部设备：

```
*port = 0x0BF4; // 写入端口
value = *port; // 从端口读取
```

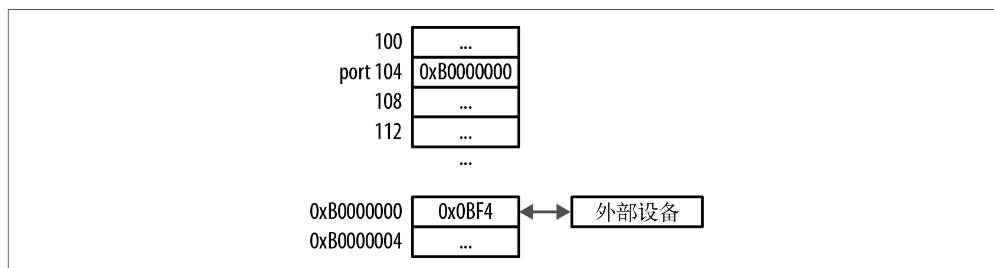
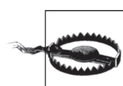


图 8-3：访问端口



用非 volatile 变量访问 volatile 内存不是个好主意，这么做会导致未定义的行为。

8.1.3 用DMA访问内存

直接内存访问（Direct Memory Access，DMA）是一种辅助系统在内存和某些设备间传输数据的底层操作，它不属于 ANSI C 规范，但是操作系统通常提供对这种操作的支持。DMA 操作一般与 CPU 并行进行，这样可以将 CPU 解放出来执行其他任务，从而得到更好的性能。

程序员先调用 DMA 函数，然后等待操作完成。通常，程序员会提供一个回调函数，当操作完成后，操作系统会调用回调函数，回调函数由函数指针指定，8.3.2 节中会深入讨论。

8.1.4 判断机器的字节序

我们可以使用类型转换操作来判断架构的字节序。字节序是指内存单元中字节的顺序，字节序一般分为小字节序和大字节序，比如说，采用小字节序表示整数的 4 个字节中的低地址用来存储整数的低位。

在下例中，我们把整数的地址从指针转换为 char，然后打印各个字节：

```
int num = 0x12345678;
char* pc = (char*) &num;
for (int i = 0; i < 4; i++) {
    printf("%p: %02x \n", pc, (unsigned char) *pc++);
}
```

这个代码片段在 Intel PC 上的输出如下，反映了这是小字节序架构。图 8-4 说明了这些值在内存中如何分配。

```
100: 78
101: 56
102: 34
103: 12
```

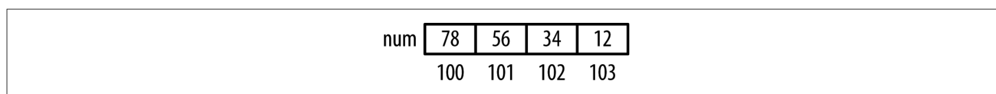


图 8-4：字节序示例

8.2 别名、强别名和restrict关键字

如果两个指针引用同一内存地址，我们称一个指针是另一个指针的别名。别名并不罕见，不过可能会引发一些问题。下面的代码声明了两个指针，并把它们都指向同一地址：

```
int num = 5;
int* p1 = &num;
int* p2 = &num;
```

当编译器为指针生成代码时，除非特别指定，它必须假设可能会存在别名。使用别名会对编译器生成代码有所限制，如果两个指针引用同一位置，那么任何一个都可能修改这个位置。当编译器生成读写这个位置的代码时，它就不能通过把值放入寄存器来优化性能。对于每次引用，它只能执行机器级别的加载和保存操作。频繁的加载/保存会很低效，在某些情况下，编译器还必须关心操作执行的顺序。

*强别名*是另一种别名，它不允许一种类型的指针成为另一种类型的指针的别名。下面的代码中，一个整数指针是一个浮点数指针的别名，这破坏了强别名的规则。这段代码判断一个数是否为负数，相比将它的参数跟 0 比较来判断正负，这种方法执行速度更快：

```
float number = 3.25f;
unsigned int *ptrValue = (unsigned int *)&number;
unsigned int result = (*ptrValue & 0x80000000) == 0;
```



强别名规则对符号或修饰符不起作用，下面都是合法的强别名：

```
int num;
const int *ptr1 = &num;
int *ptr2 = &num;
int volatile ptr3 = &num;
```

不过，有些情况下，对同样的数据采用不同的表现形式也是有用的，为了避免别名问题，可以采用这几种技术：

- 使用联合体；
- 关闭强别名；
- 使用 char 指针。

两种数据类型的联合体可以避开强别名的问题，8.2.1 节中会讨论这个主题。如果编译器有禁用强别名的选项，就可以关闭它。GCC 编译器有如下的编译器选项：

- -fno-strict-aliasing 可以关闭强别名；
- -fstrict-aliasing 可以打开强别名；
- -Wstrict-aliasing 可以打开跟强别名相关的警告信息。

需要关闭强别名的代码可能意味着差劲的内存访问实践，如果可能的话，花些时间解决这些问题，而不是关闭强别名。



编译器并非总能准确地报告别名相关的警告，有时候会漏报，有时候会虚报，最终还是要靠程序员定位别名问题。

编译器总是假定 char 指针是任意对象的潜在别名，所以，大部分情况下可以安全地使用。不过，把其他数据类型的指针转换成 char 指针，再把 char 指针转换成其他数据类型的指针，则会导致未定义的行为，应该避免这么做。

8.2.1 用联合体以多种方式表示值

C 是类型语言，在声明变量时就得为其指定类型。可以存在不同类型的多个变量，有时候，可能需要把一种类型转换成另一种类型，这一般是通过类型转换实现的，不过也可以使用联合体。*类型双关*就是指这种绕开类型系统的技术。

如果转换涉及指针，可能会产生严重问题。为了说明这种技术，我们会用到三个不同的函数，这些函数会判断一个浮点数是否为正。

第一个函数用了浮点数和无符号整数的联合体，如下所示，函数先把浮点数赋给联合体，然后再获取整数来执行测试：

```
typedef union _conversion {
    float fNum;
    unsigned int uiNum;
} Conversion;

int isPositive1(float number) {
    Conversion conversion = { .fNum =number};
    return (conversion.uiNum & 0x80000000) == 0;
}
```

这样可以正确工作，也不会涉及别名，因为没有用到指针。下面这个版本用了两种数据类型的指针的联合体，将浮点数的地址赋给第一个指针，然后解引整数指针来执行测试。这样破坏了强别名规则：

```
typedef union _conversion2 {
    float *fNum;
    unsigned int *uiNum;
} Conversion2;

int isPositive2(float number) {
    Conversion2 conversion;
    conversion.fNum =&number;
    return (*conversion.uiNum & 0x80000000) == 0;
}
```

下面这个函数没有用联合体，而且破坏了强别名规则，因为 ptrValue 指针和 number 共享了同一个地址：

```
int isPositive3(float number) {
    unsigned int *ptrValue = (unsigned int *)&number;
    return (*ptrValue & 0x80000000) == 0;
}
```

这三个函数所用的方法做了如下假设：

- 表示浮点数用的是 IEEE-754 浮点数标准；
- 以特定方式布局浮点数；
- 正确对齐了整数和浮点数指针。

不过，这些假设不一定有效。类似方法可以优化性能，但不一定可移植。如果移植性变得重要，执行浮点数比较是更好的办法。

8.2.2 强别名

编译器不会强制使用强别名，它只会产生警告。编译器假设两个或更多不同类型的指针永远不会引用同一个对象，这也包括除名字外其他都相同的结构体的指针。有了强别名，编译器可以做某些类型的优化。如果这个假设不正确，就会产生不可预期的结果。

即使两个结构体的字段完全一样，但如果名字不同的话，这两种结构体的指针就不应该引用同一对象。在下面的例子中，我们假设 person 和 employee 指针永远不会引用同一对象：

```
typedef struct _person {
    char* firstName;
```

```
    char* lastName;
    unsigned int age;
} Person;

typedef struct _employee {
    char* firstName;
    char* lastName;
    unsigned int age;
} Employee;

Person* person;
Employee* employee;
```

不过，如果定义了同一个结构体的两个类型，那么指向不同名字的指针可以引用同一对象：

```
typedef struct _person {
    char* firstName;
    char* lastName;
    unsigned int age;
} Person;

typedef Person Employee;

Person* person;
Employee* employee;
```

8.2.3 使用restrict关键字

C 编译器默认假设指针有别名，用 restrict 关键字可以在声明指针时告诉编译器这个指针没有别名，这样就允许编译器产生更高效的代码。很多情况下这是通过缓存指针实现的，不过要记住这只是个建议，编译器也可以选择不优化代码。如果用了别名，那么执行代码会导致未定义行为，编译器不会因为破坏强别名假设而提供任何警告信息。



新开发的代码应该尽量对指针声明使用 restrict 关键字，这样会产生更高效的代码，而修改已有代码可能就不划算了。

下面这个函数说明 restrict 关键字的定义和使用，该函数把两个向量相加，并将结果存在第一个向量中：

```
void add(int size, double * restrict arr1, const double * restrict arr2) {
    for (int i = 0; i < size; i++) {
        arr1[i] += arr2[i];
    }
}
```

两个指针参数都用了 restrict 关键字，但是它们不应该引用同一块内存，下面是函数的正确用法：

```
double vector1[] = {1.1, 2.2, 3.3, 4.4};
double vector2[] = {1.1, 2.2, 3.3, 4.4};

add(4,vector1,vector2);
```

在下面的代码中，两个参数用了同一个向量，这样的调用是不正确的。第一个调用语句用了别名，而第二个调用语句则使用了同一个向量两次：

```
double vector1[] = {1.1, 2.2, 3.3, 4.4};
double *vector3 = vector1;

add(4,vector1,vector3);
add(4,vector1,vector1);
```

尽管这么做有时候能正确工作，但是调用函数的结果是不可靠的。

一些标准 C 函数用了 restrict 关键字，包括：

- `void *memcpy(void * restrict s1, const void * restrict s2, size_t n);`
- `char *strcpy(char * restrict s1, const char * restrict s2);`
- `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`
- `int printf(const char * restrict format, ... );`
- `int sprintf(char * restrict s, const char * restrict format, ... );`
- `int snprintf(char * restrict s, size_t n, const char * restrict format, ... );`
- `int scanf(const char * restrict format, ...);`

restrict 关键字隐含了两层含义：

(1) 对于编译器来说，这意味着它可以执行某些代码优化；
(2) 对于程序员来说，这意味着这些指针不能有别名，否则操作的结果将是未定义的。

8.3 线程和指针

线程之间共享数据会引发一些问题。常见的问题是数据损坏。线程可以写入对象，但可能时不时地被挂起，导致对象处于不一致的状态。之后另一个线程可能会在第一个线程继续写入之前读取对象，那么第二个线程就会使用无效的或损坏的数据。

指针是在另一个线程中引用数据的常见方式，我们会研究对多线程应用程序造成不利影响的一些问题。正如我们将在本节的例子中看到的那样，很多时候会用互斥锁

保护数据。

C11 标准实现了线程，但是目前还没有得到广泛支持。有很多库可以让 C 支持线程，我们会用 POSIX 线程，因为这个库已经可用。不管用什么库，这里展示的技术都适用。

我们用指针支持多线程程序和回调函数。这里涉及线程这个主题，本章假设你熟悉基本的线程概念和术语，所以，我们不会深入讨论 POSIX 线程函数的工作原理。读者可以查看 O'Reilly 的 *Pthreads Programming*（http://shop.oreilly.com/product/9781565921153.do）获取关于这个主题的详细讨论。

8.3.1 线程间共享指针

两个或更多线程共享数据可能损坏数据。为了说明这个问题，我们会实现一个计算两个向量点积的多线程函数。多个线程会同时访问两个向量和一个和字段。当线程完成后，和字段会持有点积的值。

两个向量的点积通过把每个向量对应的元素相乘后得到的积再相加来计算。我们会用到两个数据结构支持这个操作。第一个是 `VectorInfo`，包含关于被操作向量的信息，它有两个向量的指针、`sum` 字段（持有点积），以及 `length` 字段（指定点积函数要用的向量段的长度）。`length` 字段表示线程处理的向量的部分，不是整个向量的长度：

```
typedef struct {
    double *vectorA;
    double *vectorB;
    double sum;
    int length;
} VectorInfo;
```

第二个数据结构是 `Product`，包含一个 `VectorInfo` 指针和计算点积向量的起始索引。我们会用不同的起始索引为每个线程创建这个结构体的实例：

```
typedef struct {
    VectorInfo *info;
    int beginningIndex;
} Product;
```

所有线程会在同一时间对两个向量进行计算，但是它们访问的是向量的不同部分，所以不会有冲突。每个线程会计算自己负责的那些向量的和。不过，这个和需要累加到 `VectorInfo` 结构体的 `sum` 字段上。多个线程可能会同时访问 `sum` 字段，所以需要用*互斥锁*保护数据，下面会声明互斥锁。同一时间互斥锁只允许一个线程访问受保护的变量。下面声明的互斥锁保护 `sum` 变量，我们在全局区域声明它以允许多

个线程访问：

```
pthread_mutex_t mutexSum;
```

后面会列出 dotProduct 函数，创建线程时会调用它。我们用的是 POSIX，所以需要声明这个函数的返回值为空，参数为一个 void 指针。这个指针可以给函数传递信息，这里传递的是一个 Product 结构体实例。

在函数内部，我们声明了变量来持有起始索引和结束索引。for 循环执行实际的乘法，并在 total 变量中保存累积的总和。函数的最后部分会锁住互斥锁，把 total 加到 sum 上，然后解开互斥锁。锁住互斥锁后，其他线程无法访问 sum 变量：

```
void dotProduct(void *prod) {
    Product *product = (Product*)prod;
    VectorInfo *vectorInfo = Product->info;
    int beginningIndex = Product->beginningIndex;
    int endingIndex = beginningIndex + vectorInfo->length;
    double total = 0;

    for (int i = beginningIndex; i < endingIndex; i++) {
        total += (vectorInfo->vectorA[i] * vectorInfo->vectorB[i]);
    }

    pthread_mutex_lock(&mutexSum);
    vectorInfo->sum += total;
    pthread_mutex_unlock(&mutexSum);

    pthread_exit((void*) 0);
}
```

创建线程的代码如下所示。我们声明了两个简单向量，还有一个 VectorInfo 实例，每个向量有 16 个元素，length 字段设置为 4：

```
#define NUM_THREADS 4

void threadExample() {
    VectorInfo vectorInfo;
    double vectorA[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
        9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
    double vectorB[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
        9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};

    double sum;

    vectorInfo.vectorA = vectorA;
    vectorInfo.vectorB = vectorB;
    vectorInfo.length = 4;
```

下面创建了一个 4 个元素的线程数组，还有初始化互斥锁和线程的属性字段的代码：

```
    pthread_t threads[NUM_THREADS];

    void *status;
    pthread_attr_t attr;

    pthread_mutex_init(&mutexSum, NULL);
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    int returnValue;
    int threadNumber;
```

每次迭代都会创建一个新的 `Product` 实例，我们会把 `vectorInfo` 的地址和一个基于 `threadNumber` 得到的唯一索引赋给它，然后创建线程：

```
    for (threadNumber = 0; threadNumber < NUM_THREADS; threadNumber++) {
        Product *product = (Product*) malloc(sizeof(Product));
        product->beginningIndex = threadNumber * 4;
        product->info = &vectorInfo;
        returnValue = pthread_create(&threads[threadNumber], &attr,
                            dotProduct, (void *) (void*) (product));
        if (returnValue) {
            printf("ERROR; Unable to create thread: %d\n", returnValue);
            exit(-1);
        }
    }
```

`loop` 循环结束后，销毁线程属性和互斥锁，`for` 循环确保程序等到 4 个线程都完成后打印点积。对于上面的向量，得到的是 1496：

```
        pthread_attr_destroy(&attr);

        for (int i = 0; i < NUM_THREADS; i++) {
            pthread_join(threads[i], &status);
        }

        pthread_mutex_destroy(&mutexSum);
        printf("Dot Product sum: %lf\n", vectorInfo.sum);
        pthread_exit(NULL);

    }
```

这样就可以保护 `sum` 字段。

8.3.2 用函数指针支持回调

前面我们在第 5 章中开发的 `sort` 函数用到了回调函数。因为排序示例没有用到多线程，有些程序员认为这不是回调函数。大家普遍认可的定义是如果一个线程的事件导致另一个线程的函数调用，就称为回调。将回调函数的指针传递给线程，而函数的某个事件会引发对回调函数的调用，这种方法在 GUI 应用程序中处理用户线程

事件很有用。

我们会用一个计算阶乘的函数说明这种方法，这个函数会在计算完阶乘后回调第二个函数。关于阶乘的信息封装在 FactorialData 结构体中，并在两个函数之间传递。这个结构体和阶乘函数如下所示，数据包括阶乘数、结果和回调用的函数指针。factorial 函数用这些数据计算阶乘，把结果保存到 result 字段，调用回调函数，然后结束线程：

```
typedef struct _factorialData {
    int number;
    int result;
    void (*callBack)(struct _factorialData*);
} FactorialData;

void factorial(void *args) {
    FactorialData *factorialData = (FactorialData*) args;
    void (*callBack)(FactorialData*); // 函数原型

    int number = factorialData->number;
    callBack = factorialData->callBack;

    int num = 1;
    for(int i = 1; i<=number; i++) {
        num *= i;
    }

    factorialData->result = num;
    callBack(factorialData);

    pthread_exit(NULL);
}
```

我们在 startThread 函数中创建一个线程，如下所示。这个线程会执行 factorial 函数，给它传递阶乘数据：

```
void startThread(FactorialData *data) {
    pthread_t thread_id;
    int thread = pthread_create(&thread_id, NULL, factorial, (void *) data);
}
```

回调函数只是简单地打印阶乘结果：

```
void callBackFunction(FactorialData *factorialData) {
    printf("Factorial is %d\n", factorialData->result);
}
```

初始化阶乘数据和调用 startThread 函数的代码如下所示。Sleep 函数为所有线程正常结束提供足够的时间：

```
    FactorialData *data =
        (FactorialData*) malloc(sizeof(FactorialData));

    if(!data) {
        printf("Failed to allocate memory\n");
        return;
    }

    data->number = 5;
    data->callBack = callBackFunction;

    startThread(data);
    Sleep(2000);
```

执行代码后输出如下：

```
Factorial is 120
```

程序也可以执行其他任务而不休眠，它无需等待线程完成。

8.4　面向对象技术

C 不支持面向对象编程，不过，借助不透明指针，我们也可以使用 C 封装数据以及支持某种程度的多态行为。我们可以隐藏数据结构的实现和支持函数，用户没有必要知道数据结构的实现细节，减少这些实现细节就可以降低应用程序的复杂度。此外，这样也不会引诱用户使用数据结构的内部细节，如果用户使用了，之后数据结构的实现发生变化后会导致问题。

多态行为可以帮助提高应用程序的可维护性。多态函数的行为取决于它执行的目标对象，这意味着我们可以更容易地为应用程序添加功能。

8.4.1　创建和使用不透明指针

不透明指针用来在 C 中实现数据封装。一种方法是在头文件中声明不包含任何实现细节的结构体，然后在实现文件中定义与数据结构的特定实现配合使用的函数。数据结构的用户可以看到声明和函数原型，但是实现会被隐藏（在 .c/.obj 文件中）。

只有使用数据结构所需的信息会对用户可见，如果太多的内部信息可见，用户可能会使用这些信息，从而产生依赖。一旦内部结构发生变化，用户的代码可能就会失效。

我们会开发一个链表来说明不透明指针的用法。用户会用一个函数来获取链表指针，然后用这个指针来向链表添加信息以及从链表删除信息。链表的内部结构细节和支持函数对用户不可见。这个结构的唯一可见部分通过头文件提供，如下所示：

```
//link.h

typedef void *Data;
typedef struct _linkedList LinkedList;

LinkedList* getLinkedListInstance();
void removeLinkedListInstance(LinkedList* list);
void addNode(LinkedList*, Data);
Data removeNode(LinkedList*);
```

Data 声明为 void 指针，这样允许实现处理任何类型的数据。LinkedList 的类型定义用了名为 _linkedList 的结构体，这个结构体的定义在实现文件中，对用户隐藏。

我们提供了四种方法来使用链表。用户一开始用 getLinkedListInstance 函数来获取一个 LinkedList 实例，一旦不再需要链表就应该调用 removeLinkedListInstance 函数。通过传递链表指针可以让函数处理一个或多个链表。

要将数据添加到链表，需要用 addNode 函数，我们给它传递链表和要添加到链表的数据指针。removeNode 方法会返回链表头部的数据。

链表的实现在名为 link.c 的独立文件中。实现的第一部分，如下所示，声明持有用户数据和下一个链表节点的结构体，接着是 _linkedList 结构体的定义。在这个简单的链表中，我们只用到了一个头指针：

```
// link.c

#include <stdlib.h>
#include "link.h"

typedef struct _node {
      Data* data;
      struct _node* next;
} Node;

struct _linkedList {
    Node* head;
};
```

实现文件的第二部分包含链表的四个支持函数的实现，第一个函数返回一个链表实例：

```
LinkedList* getLinkedListInstance() {
    LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
    list->head = NULL;
    return list;
}
```

接着是 removeLinkedListInstance 函数的实现，如果有节点的话，它会释放链表中的所有节点，然后释放链表本身。如果节点引用的数据包含指针，这个实现可能会产生内存泄漏。一种解决方案是传递一个释放数据成员的函数：

```
void removeLinkedListInstance(LinkedList* list) {
    Node *tmp = list->head;
    while(tmp != NULL) {
        free(tmp->data); // 潜在的内存泄漏！
        Node *current = tmp;
        tmp = tmp->next;
        free(current);
    }
    free(list);
}
```

addNode 函数把第二个参数传入的数据添加到第一个参数指定的链表中。系统会为每个节点分配内存，然后将其和用户的数据关联起来。在这个实现中，总是将链表的节点添加到头部：

```
void addNode(LinkedList* list, Data data) {
    Node *node = (Node*)malloc(sizeof(Node));
    node->data = data;
    if(list->head == NULL) {
        list->head = node;
        node->next = NULL;
    } else {
        node->next = list->head;
        list->head = node;
    }
}
```

removeNode 函数返回跟链表中第一个节点关联的数据。我们会调整头指针，让其指向链表中下个节点。接着返回数据，释放旧节点，将数据返回堆中。



用户使用这种方法无需记住释放链表节点，从而避免了内存泄漏。这是隐藏实现细节的巨大优势：

```
Data removeNode(LinkedList* list) {
    if(list->head == NULL) {
        return NULL;
    } else {
        Node* tmp = list->head;
        Data* data;
        list->head = list->head->next;
        data = tmp->data;
        free(tmp);
        return data;
    }
}
```

为了说明这个数据结构的使用方法，我们会重用 6.1 节中开发的 Person 结构体及其函数。下面的代码会把两个人添加到链表中，然后删除。首先调用 getLinkedListInstance 函数来获取链表。接着，用 initializePerson 函数创建一个 Person 实例并用 addNode 函数将其添加到链表中。displayPerson 函数会打印由 removeNode 函数返回的人。最后释放链表：

```
#include "link.h";
...
    LinkedList* list = getLinkedListInstance();

    Person *person = (Person*) malloc(sizeof(Person));
    initializePerson(person, "Peter", "Underwood", "Manager", 36);
    addNode(list, person);
    person = (Person*) malloc(sizeof(Person));
    initializePerson(person, "Sue", "Stevenson", "Developer", 28);
    addNode(list, person);

    person = removeNode(list);
    displayPerson(*person);

    person = removeNode(list);
    displayPerson(*person);

    removeLinkedListInstance(list);
```

这种方法有几个有趣的地方。我们只能在 link.c 文件中创建 _linkedList 结构体的实例，这是因为如果没有完整的结构体声明就无法使用 sizeof 操作符。比如说，如果你试图在 main 函数中为这个结构体分配内存，如下所示，会得到一个语法错误：

```
LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
```

产生的语法错误类似下面这样：

```
error: invalid application of 'sizeof' to incomplete type 'LinkedList'
```

类型不完整是因为编译器看不到 link.c 文件中的实际定义。它只能看到 _linkedList 结构体的类型定义，而看不到结构体的实现细节。

我们不允许用户看到链表内部结构以及使用链表内部结构，并且会对用户隐藏结构体的任何变化。

只有四个支持函数的签名对用户是可见的，否则，用户就无法利用或修改实现细节。我们封装了链表结构及其支持函数，从而减轻了用户的负担。

8.4.2 C中的多态

C++ 这类面向对象语言的多态是建立在基类及派生类之间继承关系的基础上的。C

不支持继承，所以我们得模拟结构体之间的继承。我们会定义和使用两个结构体来说明多态行为。Shape 结构体表示基“类”，而 Rectangle 结构体表示从基类 Shape 派生的类。

结构体的变量分配顺序对这种技术的工作原理影响很大。当我们创建一个派生类 / 结构体的实例时，会先分配基类 / 结构体的变量，然后分配派生类 / 结构体的变量。我们也需要考虑打算覆盖的函数。



理解从类实例化来的对象如何分配内存是理解面向对象语言中继承和多态工作原理的关键。我们在 C 中使用这种技术时，这一点仍然适用。

让我们先从 Shape 结构体的定义开始，如下所示。首先，我们分配一个结构体来为 Shape 结构体持有函数指针，接着是表示 x 和 y 坐标的整数：

```
typedef struct _shape {
    vFunctions functions;
    // 基类变量
    int x;
    int y;
} Shape;
```

我们将 vFunctions 结构体及其支持函数声明定义如下。当对一个类 / 结构体执行函数时，其行为取决于它所作用的对象是什么。比如说，对 Shape 调用打印函数就会显示一个 Shape，对 Rectangle 调用打印函数就会显示 Rectangle。在面向对象编程语言中这通常是通过虚表（或者 VTable）实现的。vFunctions 结构体就是用来实现这种功能的：

```
typedef void (*fptrSet)(void*,int);
typedef int (*fptrGet)(void*);
typedef void (*fptrDisplay)();

typedef struct _functions {
    // 函数
    fptrSet setX;
    fptrGet getX;
    fptrSet setY;
    fptrGet getY;
    fptrDisplay display;
} vFunctions;
```

这个结构体由一系列函数指针组成。fptrSet 和 fptrGet 函数指针为整数类型数据定义了典型的 getter 和 setter 函数。在这种情况下，它们用来获取和设置 Shape 或 Rectangle 的 x 和 y 值。fptrDisplay 函数指针定义了一个参数为空、返回值为空的函数，我们会用这个打印函数解释多态行为。

Shape 结构体有四个函数跟它协同工作，如下所示。它们的实现很直观。为了让示例简单，在 display 函数中，我们只是打印出了字符串 "Shape"，我们会把 Shape 实例作为第一个参数传递给这些函数，这样可以让这些函数处理多个 Shape 实例：

```
void shapeDisplay(Shape *shape) { printf("Shape\n");}
void shapeSetX(Shape *shape, int x) {shape->x = x;}
void shapeSetY(Shape *shape, int y) {shape->y = y;}
int shapeGetX(Shape *shape) { return shape->x;}
int shapeGetY(Shape *shape) { return shape->y;}
```

为了辅助创建 Shape 实例，我们提供了 getShapeInstance 函数，它为对象分配内存，然后为其设置函数：

```
Shape* getShapeInstance() {
    Shape *shape = (Shape*)malloc(sizeof(Shape));
    shape->functions.display = shapeDisplay;
    shape->functions.setX = shapeSetX;
    shape->functions.getX = shapeGetX;
    shape->functions.setY = shapeSetY;
    shape->functions.getY = shapeGetY;
    shape->x = 100;
    shape->y = 100;
    return shape;
}
```

下面的代码说明了这些函数的使用方法：

```
Shape *sptr = getShapeInstance();
sptr->functions.setX(sptr,35);
sptr->functions.display();
printf("%d\n", sptr->functions.getX(sptr));
```

输出如下：

```
Shape
35
```

看起来为了实现 Shape 结构体这么做有点大费周章，但是一旦我们从 Shape 派生出一个 Rectangle 结构体，就会看到这么做的强大能力。这个结构体如下所示：

```
typedef struct _rectangle {
    Shape base;
    int width;
    int height;
} Rectangle;
```

Rectangle 第一个字段的内存分配和 Shape 结构体一样，如图 8-5 所示。此外，我们还加入了两个新字段 width 和 height，来表示长方形的属性。

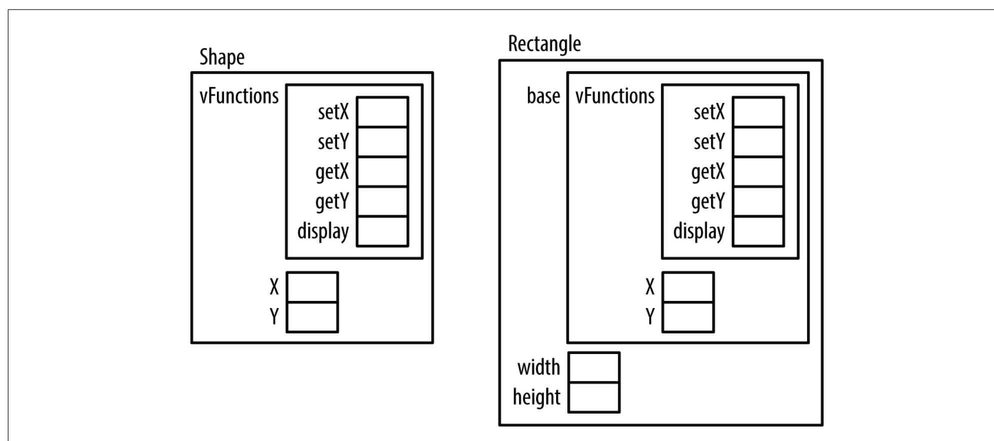


图 8-5：Shape 和 Rectangle 的内存分配

跟 Shape 类似，Rectangle 需要关联一些函数，声明如下。除了用的是 Rectangle 的 base 字段，它们跟 Shape 结构体关联的函数一样。

```
void rectangleSetX(Rectangle *rectangle, int x) {
    rectangle->base.x = x;
}

void rectangleSetY(Rectangle *rectangle, int y) {
    rectangle->base.y;
}

int rectangleGetX(Rectangle *rectangle) {
    return rectangle->base.x;
}

int rectangleGetY(Rectangle *rectangle) {
    return rectangle->base.y;
}

void rectangleDisplay() {
    printf("Rectangle\n");
}
```

getRectangleInstance 函数返回一个 Rectangle 结构体的实例，如下所示：

```
Rectangle* getRectangleInstance() {
    Rectangle *rectangle = (Rectangle*)malloc(sizeof(Rectangle));
    rectangle->base.functions.display = rectangleDisplay;
    rectangle->base.functions.setX = rectangleSetX;
    rectangle->base.functions.getX = rectangleGetX;
    rectangle->base.functions.setY = rectangleSetY;
    rectangle->base.functions.getY = rectangleGetY;
    rectangle->base.x = 200;
    rectangle->base.y = 200;
```

```
    rectangle->height = 300;
    rectangle->width = 500;
    return rectangle;
}
```

下面说明这个结构体的用法：

```
Rectangle *rptr = getRectangleInstance();
rptr->base.functions.setX(rptr,35);
rptr->base.functions.display();
printf("%d\n", rptr->base.functions.getX(rptr));
```

输出如下：

```
Rectangle
35
```

现在创建一个 Shape 指针的数组，然后像下面这样初始化。当我们把 Rectangle 赋给 shapes[1] 时，其实没有必要非得把它转换成 (Shape *)，但是不这么做会产生警告：

```
Shape *shapes[3];
shapes[0] = getShapeInstance();
shapes[0]->functions.setX(shapes[0],35);
shapes[1] = getRectangleInstance();
shapes[1]->functions.setX(shapes[1],45);
shapes[2] = getShapeInstance();
shapes[2]->functions.setX(shapes[2],55);

for(int i=0; i<3; i++) {
    shapes[i]->functions.display();
    printf("%d\n", shapes[i]->functions.getX(shapes[i]));
}
```

执行这段代码后会得到如下输出：

```
Shape
35
Rectangle
45
Shape
55
```

创建 Shape 指针的数组过程中，我们创建了一个 Rectangle 实例并将其赋给数组的第二个元素。当我们在 for 循环中打印元素时，它会用 Rectangle 的函数行为而不是 Shape 的，这就是一种多态行为。display 函数的行为取决于它所执行的对象。

我们是把它当成 Shape 来访问的，因此不应该试图用 shapes[i] 来访问其宽度和高度，原因是这个元素可能引用一个 Rectangle，也可能不是。如果我们这么做，

就可能访问 shapes 的其他内存，那些内存并不代表宽度和高度信息，会导致不可预期的结果。

现在我们也可以再从 Shape 中派生一个结构体，比如 Circle，把它加入数组，而不需要大量修改代码。我们也需要为这个结构体创建函数。

如果我们给基结构体 Shape 增加一个函数，比如 getArea，就可以为每一个类实现一个唯一的 getArea 函数。在循环中，我们可以轻易地把所有 Shape 和 Shape 派生的结构体的面积累加，而不需要先判断我们处理的是什么类型的 Shape。如果 Shape 的 getArea 实现足够了，那么我们就不需要为其他结构体增加函数了。这样很容易维护和扩展一个应用程序。

8.5　小结

本章我们探索了指针的几个方面。首先讨论了指针的类型转换，接着用示例说明了如何用指针访问内存和硬件端口，还看了如何用指针判断机器的字节序。

我们介绍了别名和 restrict 关键字。两个指针引用同一个对象就会产生别名。编译器会假设指针存在别名，不过，这可能会导致生成低效代码。restrict 关键字允许编译器执行更好的优化。

我们看到了如何结合使用指针和线程，了解了用指针共享的数据需要保护。此外，我们还研究了用函数指针在线程之间实现回调。

8.4 节研究了不透明指针和多态行为。不透明指针让 C 得以对用户隐藏数据，而应用多态的程序更容易维护。

关于作者和封面

关于作者

过去 29 年来，Richard Reese 先后服务于工业界和学术界。他曾在洛克希德·马丁公司做过 10 年的软件开发支持，那时就开发了一个基于 C 的网络应用程序。他还做过 5 年的外包讲师，为工业界提供软件培训。Richard 现在是塔尔顿州立大学副教授。

关于封面

本书封面上的动物叫做笛鸦伯劳，或者澳洲喜鹊（学名 *Cracticus tibicen*），不要跟印度尼西亚的笛鸦混淆，澳洲喜鹊根本不是乌鸦，它是伯劳鸟的亲近，原生于澳大利亚和南新几内亚。澳洲喜鹊曾经有三个独立的物种，但之后因为杂交慢慢变成了一个物种。

澳洲喜鹊的头部和身躯是黑色的，背部、双翼和尾巴上有黑白相间的羽毛。由于澳洲喜鹊能发出多种复杂的叫声，因而也被称为笛鸦伯劳。和乌鸦一样，澳洲喜鹊也是杂食鸟类，不过更喜食昆虫幼虫和无脊椎动物。它们喜欢群居，一个群落不会超过 24 只，所有群落的成员一般只会对自己的地盘进行防御性保护。不过在春天，一些繁殖后代的雄鸟会保护自己的巢穴，可能俯冲攻击经过的人及宠物。

这种喜鹊是留鸟，很容易就能适应人类的环境以及临近森林的开阔地带，因此，它不属于濒危物种。新西兰人把它当做有害物种，但是在澳大利亚，它能很好地控制外来物种甘蔗蟾蜍的数量。甘蔗蟾蜍在最初引入澳大利亚时没有天敌，有毒分泌物又助长了其数量猛增。不过，聪明的喜鹊学会了翻转甘蔗蟾蜍，戳穿其腹部，用长喙吃掉蟾蜍的内脏，这样就能避开有毒的皮肤。研究人员认为澳洲喜鹊很有希望成为甘蔗蟾蜍的天敌，帮助控制其数量。

封面上的图片来自 Wood 的 *Animate Creation*。

深入理解C指针

深入理解C指针和内存管理，提升编程效率！这是一本实战型图书，通过它，读者可以掌握指针动态操控内存的机制、对数据结构的增强支持，以及访问硬件等技术。本书详细阐述了如何在数组、字符串、结构体和函数中使用指针，同时演示了相应的内存模型及其对指针使用的影响。

指针为C语言带来了强大的功能和灵活性，却也是C语言中最难啃的一块“骨头”。本书旨在帮读者透彻理解指针，解决这个老大难问题。不论是初学者还是经验丰富的C/C++程序员和开发人员，都能从本书受益。

本书主要内容包括：

- 指针的基本概念及各种指针类型的声明；
- 学习动态内存分配、释放以及其他内存管理技术；
- 向函数传递数据和从函数返回数据；
- 理解数组和指针的关系；
- 如何通过指针使用字符串；
- 检查缓冲区溢出等指针安全问题；
- 理解不透明指针、有界指针、restrict关键字。

Richard Reese

华盛顿州塔尔顿州立大学副教授。Reese自2002年起就在塔尔顿州立大学教授计算机科学课程，此前在洛克希德•马丁公司做过十年软件开发。Reese著有多本技术图书，包括*Oracle Certified Associate*，*Java SE 7 Programmer Study Guide*，*Java 7 New Features Cookbook*，*EJB 3.1 CookBook*等。

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转604

分类建议 计算机/程序设计/C语言

人民邮电出版社网址：www.ptpress.com.cn

O'REILLY®
oreilly.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-34448-9
定价：45.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks